NPS52-88-002

NAVAL POSTGRADUATE SCHOOL

Monterey, California

INTERACTIVE, NETWORKED, MOVING
PLATFORM SIMULATORS

Michael R. Oliver
David J. Stahl, Jr.
Robert B. McGhee
Michael J. Zyda

February 1988

Approved for public release; distribution unlimited
Prepared for:
Naval Oceans Systems Center
San Diego, CA 92152

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Kneale T. Marshall
Acting Provost

This work was supported by the U.S. Army Combat Developments Experimentation Center, Fort Ord, California, the Naval Ocean Systems Center, San Diego and the Naval Postgraduate School's Direct Funding Program. This work was generated from Michael R. Oliver's and David J. Stahl, Jr.'s joint Masters Thesis.

This report was prepared by:

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-002	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Department 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION US Army Combat Developments Experiment Center Naval Ocean Systems Center
6c. ADDRESS (City, State, and ZIP Code) Naval Postgraduate School Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Ford Ord, CA 93941 San Diego, CA
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Ocean Systems Center 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MIPR ATEC 48-47 and AT E9 49-87
8c. ADDRESS (City, State, and ZIP Code) San Diego, CA	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Interactive, Networked, Moving Platform Simulators

12. PERSONAL AUTHOR(S)

Michael R. Oliver, David J. Stahl, Jr., Robert B. McGhee and Michael J. Zyda

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	February 1988	133

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Previous research has produced a real-time FOG-M missile flight simulation using Defense Mapping Agency digital terrain elevation data and a Silicon Graphics, Inc. IRIS 3120 graphics workstation. This study is a continuation of that project with the goals of providing more realistic targets and allowing viewing the terrain from inside several different types of vehicles. In addition, the use of Ethernet network communications between two workstations taking part in the simulation is used to create a missile/target gaming environment.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael J. Zyda	22b. TELEPHONE (Include Area Code) 408-646-2305 22c. OFFICE SYMBOL 522k

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-243

Interactive, Networked, Moving Platform Simulators

Michael R. Oliver, David J. Stahl, Jr., Robert B. McGhee and Michael J. Zyda *

Naval Postgraduate School,

Code 52, Dept. of Computer Science,

Monterey, California 93943

ABSTRACT

Previous research has produced a real-time FOG-M missile flight simulation using Defense Mapping Agency digital terrain elevation data and a Silicon Graphics, Inc. IRIS 3120 graphics workstation. This study is a continuation of that project with the goals of providing more realistic targets and allowing viewing the terrain from inside several different types of vehicles. In addition, the use of Ethernet network communications between two workstations taking part in the simulation is used to create a missile/target gaming environment.

‡ This work was supported by The US Army Combat Developments Experimentation Center, Fort Ord, California, the Naval Ocean Systems Center, San Diego and the Naval Postgraduate School's Direct Funding Program. This work was generated from Michael R. Oliver's and David J. Stahl, Jr.'s joint Masters Thesis.

* Contact author.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

This study is a continuation of the development of the graphics simulation described in [1]. Previous research has produced a real-time flight simulation of a missile flying over three-dimensional digitized terrain displayed on a Silicon Graphics, Inc. IRIS-3120 high performance graphics workstation. The simulation allows interactive control of the missile's speed, course, altitude and camera viewing orientation. The missile controls and camera display were designed to mimic the actual FOG-M control panel used by the military. These controls can be used to maneuver the missile over the terrain to locate, designate and destroy any target of opportunity. The targets are ten tanks arbitrarily located on the terrain map traveling at a speed of fifteen knots. Each tank's course is constant until a map boundary is reached at which time the course is reversed. Once a target has been destroyed, the program is reset allowing another missile to be launched at any of the ten original ten tank targets.

This study developed a Moving Vehicle Simulator using the same terrain database and program organization as the original FOG-M project. The vehicle simulator interfaces with the FOG-M simulator via a communication link allowing two independent users to interact with each other in real-time.

B. LIMITATIONS OF THE ORIGINAL SYSTEM

The goal of the original FOG-M study was to develop a low cost simulation of a missile flying over digitized terrain. This goal was satisfactorily achieved and has subsequently opened many new areas of research. Some of these new areas are discussed in this study. The primary goal of this study is to produce a moving vehicle simulator that presents out-the-window views from several different types of vehicles, and to incorporate both the original FOG-M system and the Moving Vehicle simulator into a network of cooperating simulators. A specific objective in continuing the original FOG-M project is to improve the simulation speed and enhance the display realism.

1. Frame Update Speed

The standard frame rate for a motion picture is twenty-four frames per second. It is the goal of any real-time program to achieve such a frame rate. The original FOG-M project has an average frame rate of three frames per second. An improved frame rate of six frames per second has been achieved by limiting slow math function calls and graphics object manipulations. The algorithm for displaying the terrain and vehicles for example, has been rewritten to draw only the polygons in the user field-of-view. While the new frame rate is still much less than that of a motion picture, it presents smooth motion.

2. Vehicle Animation

The types of vehicles that can be displayed has been changed from only tanks in the FOG-M simulator to tanks, trucks and jeeps in the Moving Vehicle simulator. These vehicles can be preset to a desired course and speed and positioned at any location on the terrain. A vehicles's course is modified when it attempts to climb a steep hill, with the

vehicle also inclining and tilting. These combined effects give a more realistic view of actual vehicles traversing rough terrain. In addition, the system allows an out-the-window view from any vehicle on the terrain.

3. Networking

To improve the realism of the project for vehicle portrayal, a second Silicon Graphics, Inc. IRIS-3120 was connected to the missile graphics workstation via an Ethernet communications link. Use of this connection allows one operator to interactively control the missile flying over the terrain, and a second operator to interactively control vehicle targets. When a vehicle is destroyed, an explosion and pile of metal is displayed.

4. User Interface

Throughout the implementation of this project, special consideration has been given to provide user friendly menus and displays. In addition, the time to load the digitized terrain data has been reduced to prevent lengthy periods of waiting.

C. ORGANIZATION

The above sections of this chapter have provided a background on the major areas described in this study. It is expected that the reader has a familiarity with computer graphics and the basics of real-time interactive computer graphics techniques. Chapter II discusses the specific efficiency improvements that have been made in developing the vehicle simulator as compared to the FOG-M system. The display improvements and additions to the FOG-M simulator such as target object creation, new data structures used and hidden surface methods employed are covered in Chapter III. The added capability of networking two workstations is discussed in Chapter IV. In addition, a brief review of

hidden surface algorithms is included with the target object discussion to give the reader a better understanding of the graphics techniques used in the simulator. Chapter V contains a user's guide for operating the Moving Vehicle Simulator. Chapter VI concludes with discussion in the areas of future recommendations for follow on research and summarizes the research conducted.

II. EFFICIENCY IMPROVEMENTS

A. PRE-FLIGHT

Pre-flight processing in the original FOG-M simulator consists of the following general steps:

- input raw terrain elevation data
- convert raw data to program internal form
- store converted data in internal storage structure
- create graphical objects for subsequent display

Data input, conversion and storage presently cause a 100 second delay before the simulator is ready to display animation. Each of these areas was examined with the intent of reducing pre-flight processing time to provide a more responsive simulator. The efficiency of the fourth area, creation of the graphical objects, is dependent on the performance of the IRIS graphics library routines, and thus was not considered.

1. Data file format

The FOG-M simulator and the moving vehicle simulator use Defense Mapping Agency (DMA) digital terrain elevation data as the source of elevation data for portraying the three-dimensional scene. This data is stored as a sequential stream of sixteen bit integers, with each two bytes representing one elevation datum. The upper three bits of this word represents the height of the vegetation at that data point. The lower thirteen bits represent the terrain elevation at that data point, without the vegetation height. The entire database is stored as shown in Figure 2-1. Data points in each square kilometer of the terrain are stored a column at a time, starting at the most western column of data. Each one kilometer length column of data is stored starting from the most

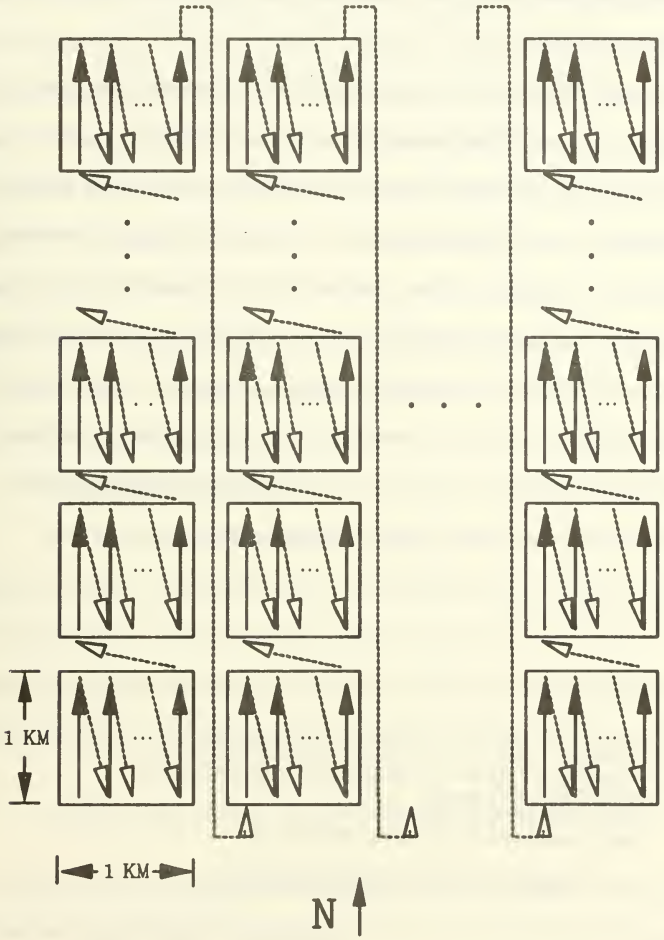


Figure 2-1 DTED File Layout

southern point in the column. The southwestern square kilometer of the database is stored first, with the remainder of the database stored in individual square kilometers: from south to north, and from west to east. A detailed description of this file layout can be found in [Ref. 1: pp. 20-24].

The original FOG-M simulator read a one-hundred square kilometer region of elevation data that was stored in the same format as the master file from which it was extracted. Elevation data points were read into a two-dimensional array of short integers, *gridpixel[][]*. Storing the set of points in this manner facilitates referencing an elevation by its X-Z coordinates. The array element *gridpixel[Z][X]* stores the elevation at north-south coordinate Z, east-west coordinate X. Due to the peculiar storage format of the DMA data, however, the DMA file must be read with a number of nested loops to store an elevation data point at the correct array index. This nesting of loops contributes to the slow pre-flight processing time readily obvious in the FOG-M simulator. A section of this FOG-M code to read the terrain elevation data appears in Figure 2-2.

```
for (coloffset = 0; coloffset < NUMXGRIDS * 10; coloffset += 10)
 for (rowoffset = 0; rowoffset < NUMZGRIDS * 10; rowoffset += 10)
  for (col = 0; col < 10; ++col)
   for (row = 0; row < 10; ++row)
    read(fd,&gridpixel[rowoffset+row][coloffset+col],2);
```

Figure 2-2. Terrain Elevation Data Input

An improvement was achieved by reformatting the terrain elevation data file to match that of the two-dimensional array in which it is stored during program execution. The reformatted file is stored as shown in Figure 2-3. Data points for ten lengths of ten kilometers are stored a row at a time, from west to east along a row's length, and from south to north, going from row to row. This matches the C compiler storage mapping function for two-dimensional arrays. An array *dted[Z_DATA_PTS][X_DATA_PTS]* is stored in memory a row at a time, starting from *dted[0][0]* through *dted[0][X_DATA_PTS-1]* for the first row, and so on for subsequent rows. To input the terrain elevation data upon program startup a single loop is executed, with an entire row of the array read at each pass through the loop. This much simplified code for reading the terrain elevation data appears in Figure 2-4.

2. Terrain Polygon Construction

The FOG-M simulator constructs a three-dimensional contour from colored triangular polygons. The ten kilometer by ten kilometer area of missile flight is sectioned into hundred meter squares, with each square consisting of two triangles. Figure 2-5 depicts this arrangement, and the terminology used. The world coordinates of triangle vertices are stored in a five dimensional array *gridcoord*. Indices of this array are:

gridcoord [Z][X][which_triangle][which_vertex][which_coordinate]

The example vertex in Figure 2-5 is located in the upper triangle at row 99, column 1, and its X, Y, and Z coordinates are stored in

gridcoord [99] [1] [U] [0] [X]
gridcoord [99] [1] [U] [0] [Y]
gridcoord [99] [1] [U] [0] [Z]

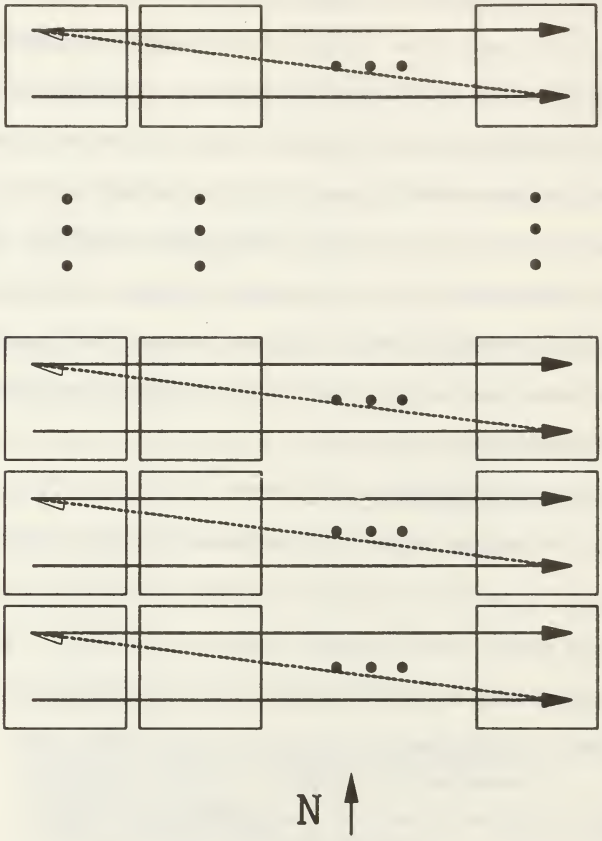


Figure 2-3 Simulator Elevation File Layout

```
for (row = 0; row < Z_DATA_PTS; ++row)
read(fd,&dted[row][0],202);
```

Figure 2-4. Terrain Elevation Data Input (Revised)

Note that these coordinates are the same for all three triangles having this common vertex. Displaying one video frame of terrain consists of looping through X and Z indices of the *gridcoord* array to select triangle coordinates for polygons to be drawn, then calling the IRIS graphics library polygon fill routine with the appropriate color. Values for the triangles' coordinates are determined prior to missile flight in the function *maketerrain()*. The elevation data array provides the height (Y) coordinate value, and a call to function *lightorient()* provides the polygon's color. As Ref [1] stated, raw elevation values are scaled to provide realism, using an exponential scaling. This requires a math library function call to the procedure *pow()*. The original FOG-M program treated each triangle's set of coordinates separately from adjacent triangles when performing this scaling, even though most vertices are shared by as many as six adjacent triangles. This resulted in a call to *pow()* six times for the same vertex, as each of the six triangles sharing the vertex were processed. A pseudo-code summary of the calls to create the terrain polygons is given in Figure 2-6.

A marked improvement in pre-flight processing time was achieved by performing the height scaling calculation only once for each triangle vertex. Once this was done, an additional improvement was realized by storing the elevation data file itself with scaling already performed on each data point. Elevation data input and storage of

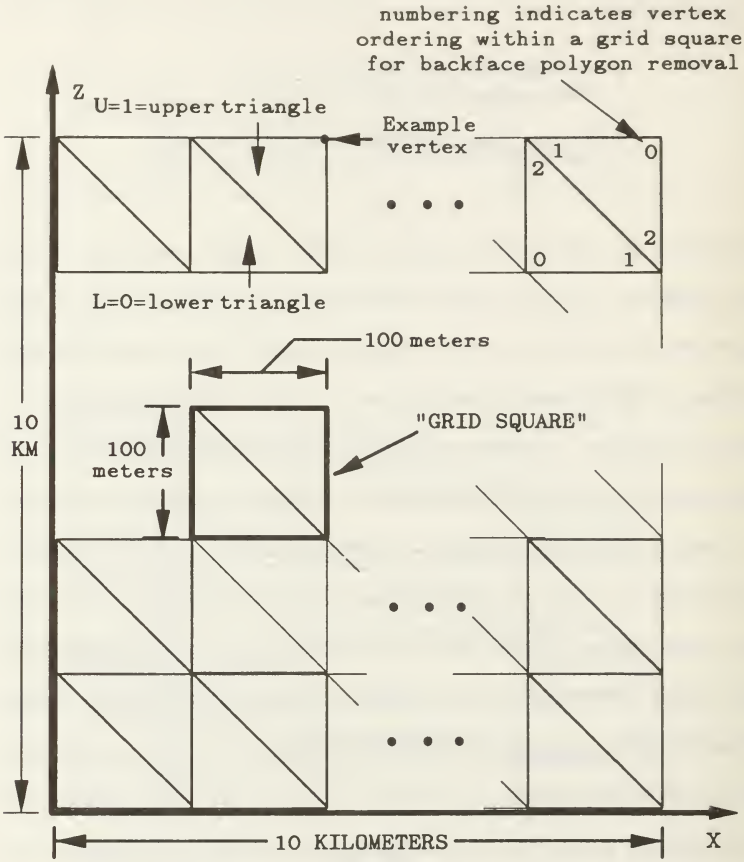


Figure 2-5 Terrain Polygons

```
maketerrain()
{
 for each row of grid squares do
  for each column of grid squares do
   for each triangle in a grid square do
    for each vertex in a triangle do
     call pow() to scale elevation (Y coordinate)
     store vertex coordinates for this triangle
     call lightorient() to determine color
}
```

Figure 2-6. Terrain Polygon Construction

terrain polygon vertex coordinates was reduced to simply reading from a file and assigning the *gridcoord* array element values, with no math library calls for scaling needed during program execution. Note however, that this requires use of an external program to properly read and scale data from the terrain master file and to properly format output to a file used by the simulator. A slight modification was also made to function *maketerrain()* to speed up determining a terrain polygon's color. Results of a run-time profiler revealed function *lightorient()* contributing most to the slow execution speed of *maketerrain()*. A change was made to *maketerrain()* that allowed the creation of a file to store polygon colors if desired. Subsequent runs of the program can then make use of the color values in this file, eliminating calls to *lightorient*. Figure 2-7 provides a pseudo-code summary of revised function *maketerrain()*.

3. Coordinate System

The initial choice of world coordinate system was based on the DMA terrain data: elevation heights in the database are measured in feet. As described above, the missile flight area is sectioned into a grid of hundred meter squares, with one hundred of

```
maketerrain()
{
 if color_file exists, read in all polygon colors
 for each row of grid squares do
  for each vertex in a row do
    store vertex coordinates for all triangles sharing this vertex
    if color_file does not exist,
      call lightorient() to determine color
      store color in color_file
}
```

Figure 2-7. Terrain Polygon Construction (Revised)

these squares in both north-south and east-west directions. This choice of using a metric grid and using data values in the grid measured in feet required several of the program's routines to perform conversions between coordinate systems. This conversion introduces costly floating point divisions and multiplications, which directly affect both pre-flight processing time and display loop run time. Since points in the terrain database file were to be scaled prior to reading the file, an additional conversion of the scaled value to its metric coordinate system equivalent value could be done to avoid the need for conversion between coordinate systems. This off-line preprocessing of the terrain database was done. In addition, a metric coordinate system was consistently assumed for all calculations in the follow-on version of the Moving Vehicle Simulator.

B. DISPLAY LOOP

Realism in an animated display depends heavily on the appearance of smooth motion. For a simulator such as the FOG-M, which constructs the scene as a collection of filled polygons, it is desired that most of the program's execution time be spent in

drawing the scene. A large portion of the display loop in the original FOG-M simulator was spent preparing the graphical objects and the data structures that manipulated them for subsequent drawing. The efficiency of the functions used in display loop calculations, and the data structures used by the program to manage the display, were examined with the intent of increasing the simulator frame rate.

1. Revised Functions

a. Ground Level

The graphics commands used to draw the three-dimensional vehicle images of the FOG-M simulator were collected into graphical objects (Chapter III gives a detailed account of the actual construction of each of these vehicle objects). Rotations and translations used to transform a typical vehicle to its correct position and orientation in the viewing volume are performed on the object as a whole. Translation of a vehicle object to the appropriate height on the terrain requires determining an interpolated height value with a call to the FOG-M program function *ground_level()*. Interpolation is necessary since height values are explicitly specified only at terrain polygon vertices, yet a vehicle can be located anywhere on the terrain. Simplification of the original *ground_level()* function resulted in an improvement of 50% in execution time for this function. Since this function is called once for every vehicle drawn in the display loop, this improvement is significant for systems having a large number of vehicles.

b. View Bounds

The Silicon Graphics IRIS workstation uses custom VLSI chips to provide hardware clipping and matrix transformations. Viewing, modeling, projection and display device transformations are performed in this high-speed pipelined architecture at

a much faster rate than is possible in software. Application programs need only specify the desired viewing volume and need not worry about clipping points, lines, planes, or surfaces to this volume. Any drawing done by an application program that takes place outside of the currently specified viewing volume is automatically clipped.

The FOG-M simulator defines a perspective viewing volume with a viewpoint located at the current position of the missile camera in world coordinates. The Moving Vehicle simulator defines a similar volume, with the viewpoint being that of the driver in the vehicle currently being operated. In both simulators, the field-of-view is limited to a maximum of fifty-five degrees. This arrangement is shown in Figure 2-8.

Depending on the location of the missile or vehicle in the viewing volume, some of the polygons in the scene are outside the field-of-view, and hence should not be seen. The simulator could take advantage of the IRIS graphics hardware clipping capability to eliminate the non-visible polygons. Conceivably, each display frame of the three-dimensional terrain contour could be generated by drawing all of the filled terrain polygons on each pass through the display loop. This is not done, simply due to the large number of polygons that comprise the terrain. With hundred meter squares, composed of two triangles each, the full one hundred square kilometer flight area contains twenty thousand polygons! Even with fast custom hardware to do the clipping, the frame rate would be too slow to provide a realistic sensation of motion over the ground. Sending only a portion of the polygons through the hardware pipeline to be clipped and drawn obviously speeds up the frame-to-frame display. It is readily apparent that none of the polygons in the direction opposite the line-of-sight are visible. Based on this fact, the original FOG-M simulator performs a determination in the function *viewbounds()* of

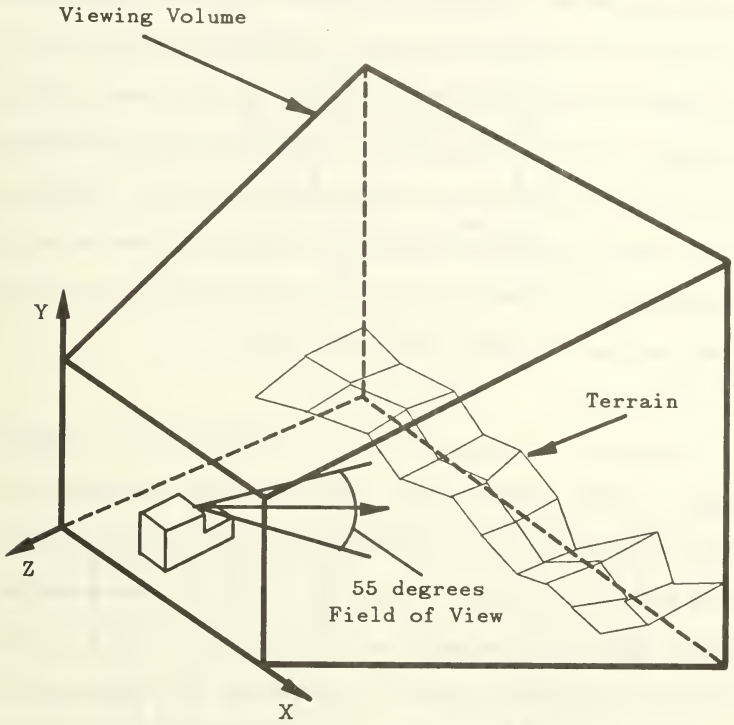
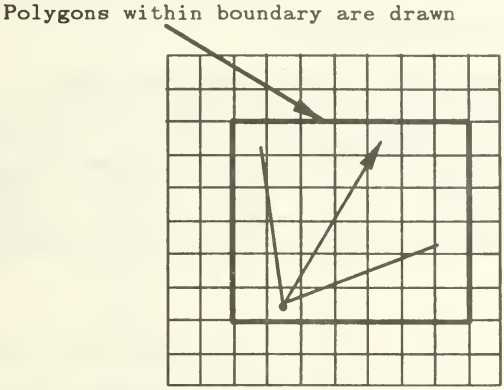


Figure 2-8 Perspective Viewing Volume

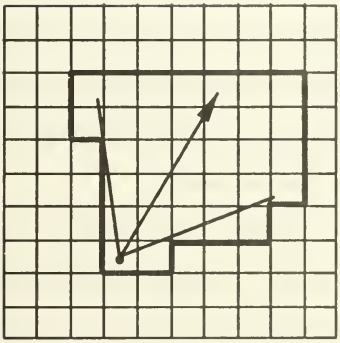
which subset of polygons to present to the clipping hardware. The method used in *viewbounds()* to do this determination results in the set of polygons to be clipped and drawn forming a rectangle around the viewpoint. However, this set still contains polygons that are not visible. Only those polygons in the line-of-sight that are actually within the field-of-view are visible. The simulator's display frame rate was increased further by revising *viewbounds()* such that only polygons in the field-of-view are actually clipped and drawn. This adds an additional benefit when the missile operator zooms the camera in: as fewer polygons are drawn, the display appears smoother. As shown in Figure 2-9, the number of polygons that are transformed, clipped, and drawn can greatly vary. Drawing only those polygons that are in the field-of-view, however, ensures fewer polygons are drawn than in the original version of the simulator.

c. Miscellaneous

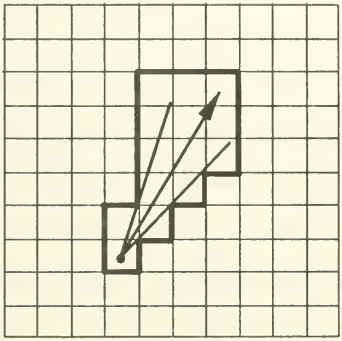
Animation of the missile and vehicles in the display is done by drawing these objects at slightly different locations from frame-to-frame. Calculations are performed in the display loop to update the position of the missile and target vehicles as they traverse the terrain. An object's new location is a function of its current position, the direction in which it is traveling and its speed. Trigonometric functions in the math library were used in the original simulator to determine new object positions and to perform calculations based on which direction the missile camera was turned. To minimize the time it takes to calculate trigonometric function values, lookup tables were constructed for the cosine and tangent functions. These tables provide quick lookup results at a resolution of one-tenth of a degree. To speed up calculations involving the arcsin relation the small angle approximation is also used. For angles less than fifteen



Original Method



Wide Field of View



Narrow Field of View

Figure 2-9 Terrain Polygons Drawn

degrees, the sine of the angle is equal to the angle itself, with an error of about one percent. This approximation is used in the Moving Vehicle simulator since arcsin values are required for small angles only. The improvement achieved using look-up tables and the small angle approximation is shown in Table 2-1.

TABLE 2-1. SPEEDING UP TRIGONOMETRIC FUNCTIONS

Function	Math Library	Approximation	Improvement
	Millisec per call		
tan	0.125	0.027	463%
sin/cos	0.259	0.049	528%
arcsin	0.157	0.011	1442%

2. Data Structures

The average frame update rate achieved by the original FOG-M simulator, which allowed a fixed number of rudimentary vehicles in the scene, is less than three frames per second. With the real vehicle dynamics capability described in Chapter III added to the simulator, the frame update rate varies with the number of vehicles drawn. Figure 2-10 shows this relationship. Even with only a small number of vehicles in the scene, the performance of the simulator degrades to an unacceptable level. The UNIX *profile* utility was used to determine exactly which routines cause this "bottleneck", with the top four time consuming display loop routines shown in Table 2-2.

The first two entries in the table are easily explained. Since the simulator makes heavy use of polygon fill to draw the desired scene, it is expected that the graphics library function *polf()* would take a considerable amount of the CPU time. Likewise, since the majority of display loop drawing takes place in function *display_terrain()*, the same conclusion can be reached. The last two entries in the table, however, are a direct result of the choice of data structure used in the original FOG-M simulator. Details of

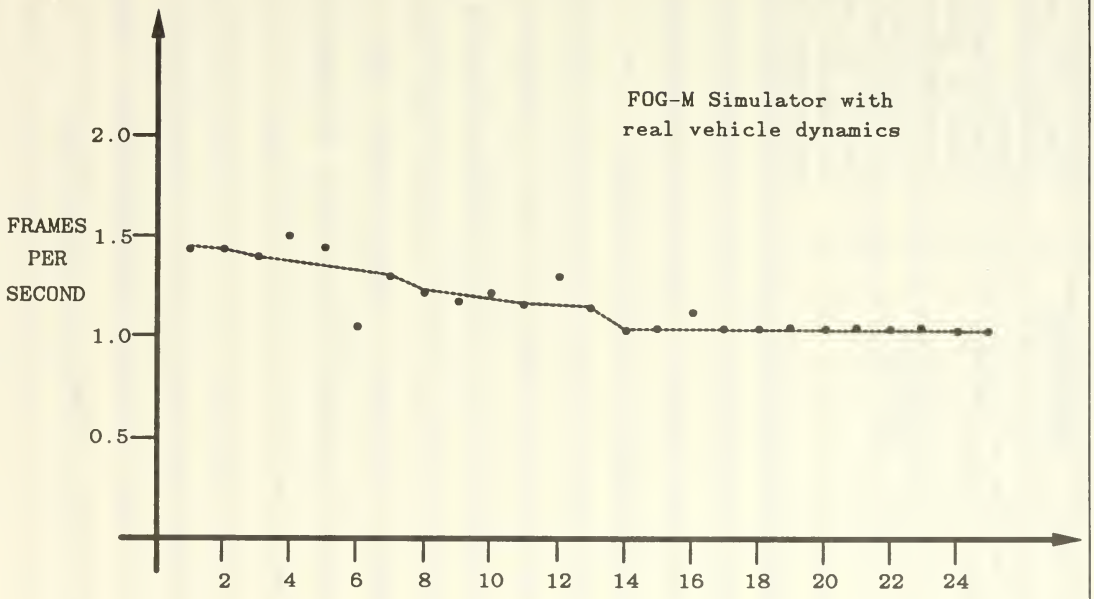


Figure 2-10 Display Rate vs Number of Vehicles (Old Data Structure)

TABLE 2-2. FOG-M ROUTINES USING THE MOST CPU TIME

% CPU Time	Routine Name	Purpose
16.9	polf	Iris graphics library filled polygon routine.
13.7	display_terrain	Output 3-D scene with hidden surface removal.
8.7	malloc	C language built in routine for dynamic memory allocation.
4.5	gl_findhash	Low level Iris graphics library routine, used for the hash tables associated with graphical objects (Not user accessible).

this implementation can be found in [Ref. 1: pp. 76-81]. A short summary of the data structures and their use in the original FOG-M simulator is presented here as background. Their impact on simulator performance was explored in this study.

The "painter's algorithm" for hidden surface elimination as described in Chapter III is used by the FOG-M simulator. This algorithm draws a scene much as a painter would, with distant objects drawn first, and with hidden surfaces overpainted by closer objects. The algorithm is easily implemented for a scene drawn as a grid of squares. For missile flight over bare terrain, without trees, buildings, or vehicles present, no other algorithm or refinement of the painter's algorithm is needed. The algorithm ensures hidden terrain surfaces are obscured by surfaces closer to the viewpoint. With nothing more in the scene than terrain polygons, there are no other surfaces that might be obscured by, or that might themselves obscure the terrain. Integration of targets into the scene introduced new complexity to the hidden surface removal problem. Management of vehicle targets in the display was attempted in the following manner.

A vehicle object moving over the terrain is associated with an element of a global two-dimensional array, with one array defined for each vehicle type. The range of indices in this array corresponds to the number of hundred meter grid squares in each dimension of the missile flight area. Tank objects for example, are associated with the array *tanks[100][100]*. The specific array element indices a particular tank is associated with is determined by the grid square it occupies. This is illustrated in Figure 2-11. Tank 'A' is situated in row Z and column X, and is associated with array element *target[Z][X]*. Similarly, tank 'B' is located in a grid square at row Z, column X+2, and is associated with array element *target[Z][X+2]*. The values stored in these arrays are the integer names of the graphical objects that should be drawn at some point in the painter's algorithm. These values are initially set to zero, indicating no vehicles are present. Once the target vehicles are defined, drawing vehicle objects on the terrain can be done by first drawing a grid square, then accessing the object name array to draw any vehicles that might be present in that grid square. Note that two or more vehicles present in one grid square are associated with the same array element, and that the commands necessary to draw these vehicles are collected into the same graphical object. In addition, a vehicle crossing grid square edges is drawn in each grid square it occupies. This results in an individual vehicle having as many as four sets of identical drawing commands in four different graphical objects to draw that vehicle correctly with respect to hidden surfaces.

Since vehicles can move from one grid square to another between frames of the display, a means was needed to reflect the changing association between vehicles and the array element they corresponded to, as used by the painter's algorithm. The choice was made to *delete* all vehicle objects and recreate them *at each pass through the display*

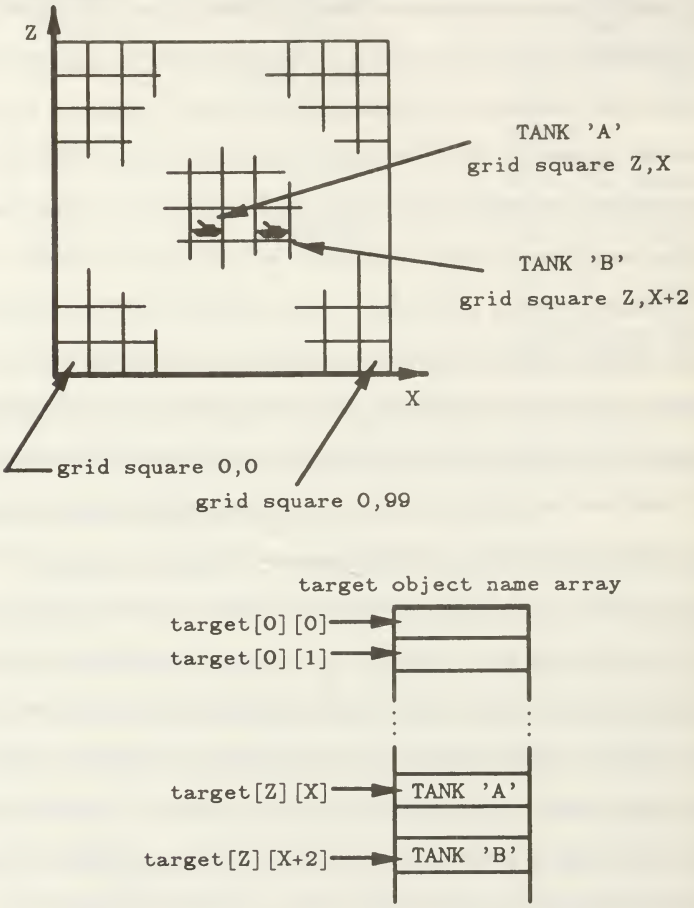


Figure 2-11 Vehicle Object Arrays

loop. This implementation is deficient in three respects. First, the repeated creation and deletion of graphical objects makes use of routines *malloc()* and *gl_findhash()*. As evidenced in Table 2-2, these two functions take a significant amount of CPU time and are the cause of the display update bottleneck. An alternative to this approach that does not make use of these slow functions is thus suggested. Second, drawing a vehicle as many as four times as it crosses a grid square edge is both time and space consuming. Third, the hidden object removal problem for vehicles in the scene is solved only for the special case where no more than one vehicle occupies each grid square. In the case of several vehicles occupying the same grid square, the commands to draw each of these objects are added to the existing object in the order the vehicles are processed, not in depth order. This results in vehicles further from the viewpoint possibly being drawn after and overwriting closer vehicles in that grid square. All of these deficiencies were corrected with choice of a different data structure for managing the display. The Chapter III discussion of hidden surface removal includes a description of this data structure. The repetitive creation and deletion of graphical objects and the need to draw vehicle objects more than once was eliminated by using the new structure. The correction of these three deficiencies has increased the simulator display rate.

C. RESULTS

Significant improvements were achieved in simulator performance using the techniques described above. Pre-processing time was considerably reduced, as shown in Table 2-3.

TABLE 2-3. PRE-FLIGHT PROCESSING TIME IMPROVEMENT

Simulator Version	Total Time
Original	1 min 41 seconds
Revised	17 seconds

The simulator frame update rate was increased by approximately a factor of three. Although this figure seems low, this improvement was achieved with the addition of real vehicle dynamics capabilities, and with the correct display of hidden surfaces in the scene. Figure 2-12 shows the frame update rates achieved in the revised simulator using the new data structures, for various number of vehicles in the scene. A UNIX *profile* indicates the success of using the new data structure. Table 2-4 lists the top four routines using the most CPU time in the revised simulator. The majority of processing time is spent in function *display_terrain()*, drawing the polygons that comprise the terrain and the vehicle objects. Table 2-5 summarizes display update rate differences between the two versions of the simulator. In this table, 'static' refers to the type of vehicle objects drawn in the original simulator. 'Dynamic' refers to vehicle objects that more accurately model normal vehicle motion over rough terrain, a feature not present in the original version of the simulator. This added capability is further explained in Chapter III. The frame rate values are for the indicated number of vehicles, the maximum fifty-five degrees field-of-view, and the largest possible number of polygons drawn, giving worst-case update rate values.

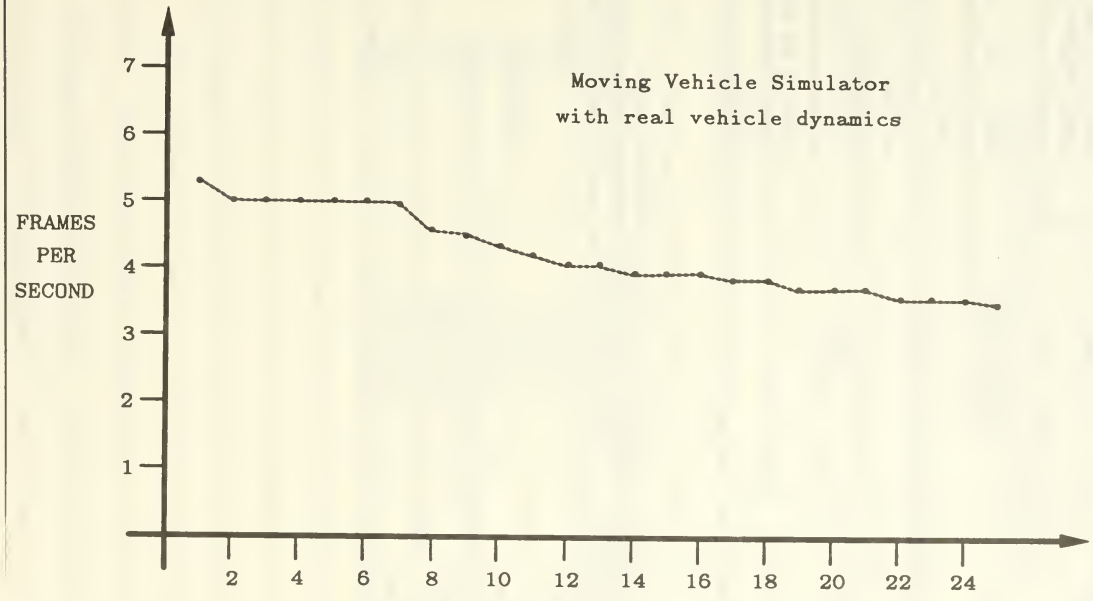


Figure 2-12 Display Rate vs. Number of Vehicles (New Data Structure)

TABLE 2-4. MOVING VEHICLE SIMULATOR ROUTINES USING THE MOST CPU TIME

% CPU Time	Routine Name	Purpose
20.2	i_polf	Low-level filled polygon primitive used by the Geometry Engine.
11.7	display_terrain	Output 3-D scene with hidden surface removal.
7.1	polf	Iris graphics library user-level filled polygon routine.
4.1	qtest	Tests for events on the valuator queue. Called in the display loop to test for menu selections.

TABLE 2-5. DISPLAY UPDATE RATE IMPROVEMENT

Simulator Version	Number of Vehicles	Frame Rate (frames/sec)
Original	1 (static)	2.6
	10 (static)	1.9
Revised	1 (static)	5.7
	10 (static)	4.0
Original	1 (dynamic)	1.4
	10 (dynamic)	1.2
Revised	1 (dynamic)	5.3
	10 (dynamic)	4.3

III. MOVING VEHICLE CONSTRUCTION AND DISPLAY

A. THREE-DIMENSIONAL GRAPHICAL DISPLAY

Many different algorithms were studied for optimizing the display of graphical objects in a three-dimensional scene. The major problem with time efficient display of graphical objects is the drawing order of the polygons required to show a non-distorted view. To solve this problem several algorithms were examined. A brief discussion of each of the drawing algorithms' merits and downfalls is given below for their implementation in a real-time graphics display. Throughout the following discussion, the term *distortion* implies an incorrect drawing order of polygons resulting in an undesired view of an object.

1. Z-Buffering

Z-Buffering is a simple yet time intensive approach to eliminate hidden surfaces [2]. This technique draws only the pixel having the smallest z position of all the polygons displayed in the viewing volume. Figure 3-1 shows two polygons A and B each having a different depth z from the view position. Since polygon A has the smallest z coordinate for the pixel point selected, its pixel is drawn instead of polygon B's pixel. Note that this comparison must be performed for each pixel of each polygon drawn.

The actual implementation of the Z-Buffering algorithm requires the use of two buffers, the z buffer for the smallest z position of all the polygons and the frame buffer for the intensity values of the closest pixel. The algorithm first initializes the buffers, then for each pixel of every polygon in the scene calculates a z coordinate and

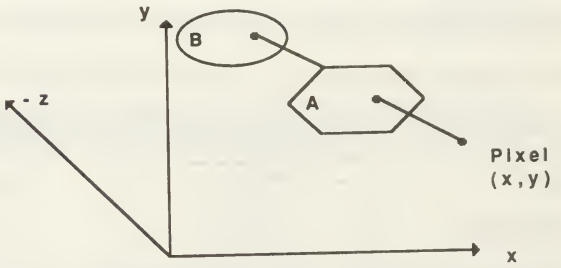


Figure 3-1. Z-Buffer Algorithm

compares it to the last z coordinate stored in the z buffer. If the new z coordinate is smaller than the z buffer coordinate, the z buffer is updated with the new value and the intensity value of the new pixel is stored in the frame buffer. After this process has been performed on all the polygons, the two buffers contain the polygon pixels and intensity values for the scene to be displayed.

2. Binary Space Partitioning

The Binary Space Partitioning (BSP) algorithm is based on storing polygons in the view volume in a sorted tree [3]. The tree is sorted with respect to a polygon being in front or back of a defined partitioning plane. To view the scene, the polygons are drawn utilizing the current viewpoint and view direction as guides to the tree's traversal.

The most difficult thing to understand about the BSP construct is how the tree is used to draw a scene. This can be explained by using a simple illustration. Take for example two halves of a block cut diagonally and separated by a small distance (Figure 3-2(a)). The algorithm takes each of the surfaces of the two halves and constructs a tree (Figure 3-2(b)). The BSP tree contains the position of all the surfaces, based on their relative location to the partitioning plane. To display a scene, a tree traversal is performed based on the viewer's position and line-of-sight.

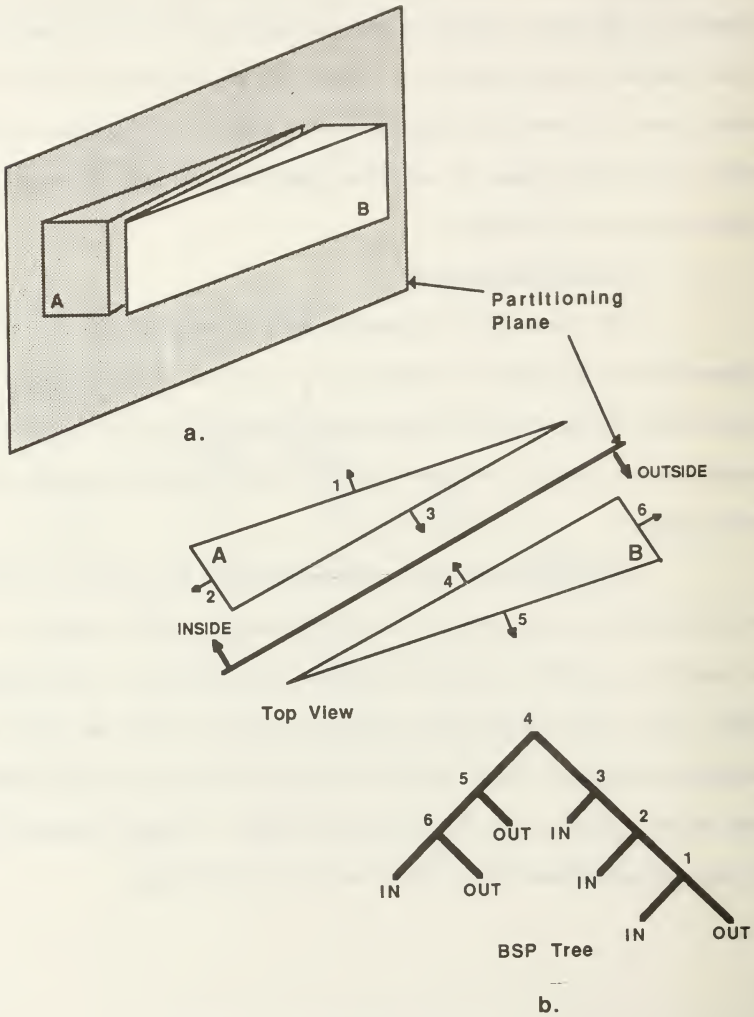


Figure 3-2. BSP Tree Construction

3. Painter's Algorithm

The painter's algorithm is related to painting a picture on canvas. A scene is created by painting the background first, followed by painting all the other objects in the scene over the background or each other based on their depth of field in the scene. In a graphics environment, this is similar to painting the furthest polygons first followed by the closest polygons. Figure 3-3 shows a progression of three polygons A, B and C with A being the closest, C the furthest and B in the middle. The painter's algorithm calculates the distance from the viewer for each polygon and draws them in order of C then B then A. Any overlapping of the polygons is obscured by the closest polygon.

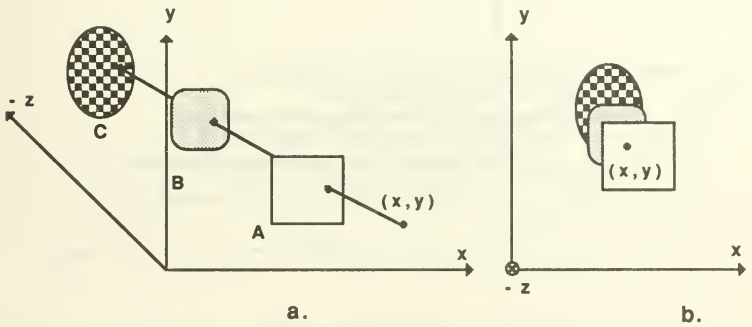


Figure 3-3. Painter's Algorithm

4. Scan Lines

The scan line algorithm is primarily used to fill polygons that are defined in a discrete order. As a scan line is defined, all the polygons that it intersects are drawn in sequence. The scan lines start at the furthest boundary of the viewing volume and are incremented towards the viewpoint one line at a time. The scan lines are produced by scanning from left to right, far to near, to draw the numbered blocks in numerical order. The scene with each scan is drawn from far to near, therefore any polygons that are closer to the viewpoint are painted over the farthest polygons (Figure 3-4). The scan line algorithm can be tailored by the designer to start at any depth and draw polygons only in the defined viewport to provide an extremely fast screen refresh rate.

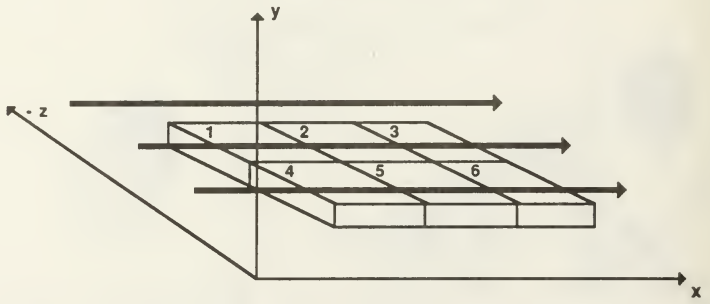


Figure 3-4. Scan Line Algorithm

5. Backface Polygon Removal

In addition to the painter's algorithm, backface polygon removal is used to draw only one side of a polygon. The backface polygon removal algorithm samples the rotation direction that the polygon points are drawn, clockwise (CW) or counter clockwise (CCW). If the polygon points are defined in a CCW rotation, that side is drawn. This drawing technique can explained by drawing a three-dimensional box (Figure 3-5). Each side of the box is drawn using backface removal to only draw the outside of the box surface. When the box is completed, all the individual sides are painted in order of depth. However, the opposite side polygon does not paint over the nearest side because its backface is not drawn.

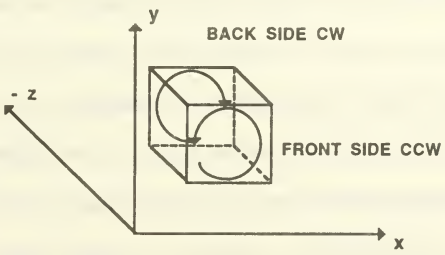


Figure 3-5. Backface Polygon Removal

B. HIDDEN SURFACE COMPARISONS

The performance of a hidden surface display method is dependent on the application environment in which it is to be used. If polygons are ordered by depth in the z direction, with minimal overlapping, a depth sorting method may be best. For polygons that are ordered in the y direction, a scan line method is best. The method employed is therefore dependent on the application.

The BSP algorithm is primarily used to display a static scene viewed from various orientations. Once the BSP tree has been constructed, the polygons are drawn by traversing the tree using the viewing position and orientation. If there is any relative motion between objects in the tree, the viewing order changes and requires a reconstruction of the tree. Since BSP tree construction is a time intensive operation, its implementation in a real-time dynamic environment is not efficient enough to refresh the display at a reasonably fast frame rate.

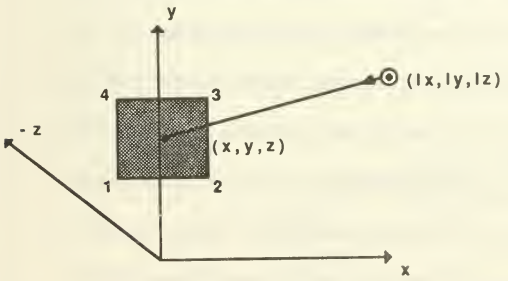
From the previous discussion, it can be seen that the Z-Buffering algorithm is easy to implement. However the Z-Buffer algorithm requires the performance of many coordinate comparisons to derive the drawing order. This method requires the use of special hardware in order to be performed at a fast frame rate. On most graphic workstations presently available, the Z-Buffer polygon fill rate is many times less that of the normal shaded polygon fill rate, making this method impractical for a large scene.

A scan line algorithm can be tailored to a dynamic scene if the number of polygons are ordered in a grid plane. Such an algorithm allows a refresh rate rapid enough to support real-time visual simulation. The designer of the algorithm needs to be able to rapidly compute the scan line ordering.

C. TARGET TYPES AS OBJECTS

The use of objects in a graphical environment is similar to a call to a programming language subroutine. An object consists of a sequence of graphics commands that are used more than once each scene. By using objects, construction time overhead can be avoided. We build our objects outside the display loop and then call those objects via a named reference [4].

The targets and missile are created as graphical objects. They are all constructed with the painter's algorithm and backface polygon removal in mind for hidden surface removal. Each polygon is drawn by defining its vertices, determining its light-shaded color, and then drawing the polygon using a graphics call to a polygon fill function (Figure 3-6). A detailed description of how each of the targets and missile are drawn is given below.




```
P1 (x) ; P1 (y); P1 (z);
P2 (x) ; P2 (y); P2 (z);
P3 (x) ; P3 (y); P3 (z);
P4 (x) ; P4 (y); P4 (z);

light_orient(lx,ly,lz,x,y,z,&shade);

color(shade);

polf(P,points);
```

Figure 3-6. Polygon Draw Sequence

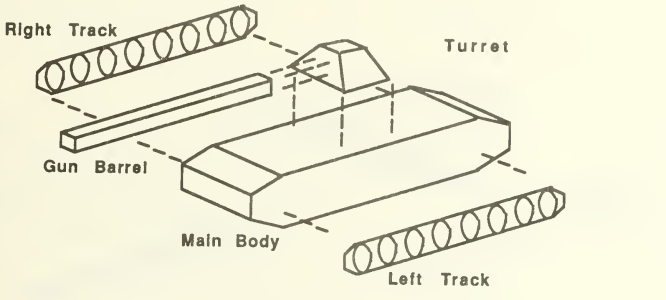
All of the objects drawn in this study are built using backface polygon removal and the painter's algorithm to provide a correct view from any point around the object from the ground plane and above. Some of the drawing techniques discussed in the subsequent paragraphs do not work if an object must be viewed from below the ground plane. The local sorting of polygons using the view direction/line-of-sight were tried in addition to the painter's algorithm. It was found that the sorting of the polygons each time the scene changed was not performed fast enough to support a real-time frame rate. Therefore, special drawing techniques are used in this study for each object.

1. Tank

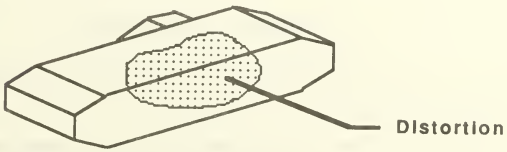
The tank used in the existing FOG-M simulator consisted of three separate parts: a turret, a main body and a gun barrel. These parts were not drawn in an order that presents an undistorted view of the tank from all directions available in three space.

A new tank object was drawn using all the same parts described above with the addition of two tank tracks. Unlike the existing tank, the new tank object polygon parts were separated and drawn in an order to always provide a realistic view (Figure 3-7(a)). Note that the order of fitting the different parts of the tank together can change the way they are drawn on the screen. For example, assume that the turret is drawn first followed by the main body. Using the painter's algorithm, this would first draw all the sides of the turret, then draw the top of the main body over the turret (Figure 3-7(b)). If we view the tank from above, similar to the view of a missile, we see a distorted tank. The solution is to draw the main body first, then the turret (Figure 3-7(c)).

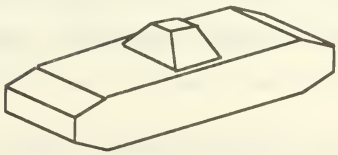
In addition to the ordering of separate parts, picture distortion can still occur (Figure 3-8). This distortion was caused by the gun barrel being drawn first,



a.



b.



c.

Figure 3-7. Tank Parts

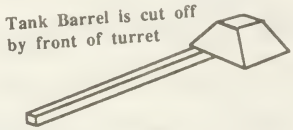


Figure 3-8. Distorted Tank

followed by the turret. The front polygon of the turret, when viewed from the front, paints over a section of the gun barrel. This problem is corrected by drawing the gun barrel with the polygons of the turret. The front of the turret is drawn first followed by the gun barrel, then the rest of the turret. This paints the gun barrel over the turret when the tank is viewed from above or the side. Now the tank is displayed without any distortion (Figure 3-9).

Drawing the tank tracks presented some new hidden surface problems. A realistic three-dimensional view of the tank with tracks cannot be achieved by using a simple drawing order. The tank tracks are created as a separate object named *track*, and

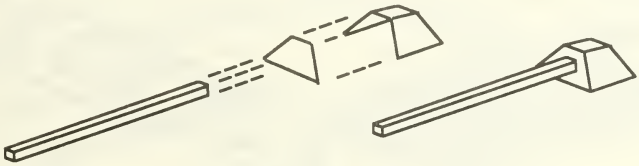


Figure 3-9. Tank Turret and Gun Drawing Order

are translated and drawn after the tank's main body (Figure 3-10(a)). This drawing order would normally not cause a display distortion if a simple track object was used. Unfortunately the tank track is not a simple object (Figure 3-10 (b)).

To maintain a realistic tank image, the track had to also be a three dimensional object with four sides. Therefore the drawing order of

main body, right side
right track
main body, left side
left track

is used, with only a few translations to maintain high drawing efficiency. This drawing order is only distorted when viewing the tank from the right side. The distortion is

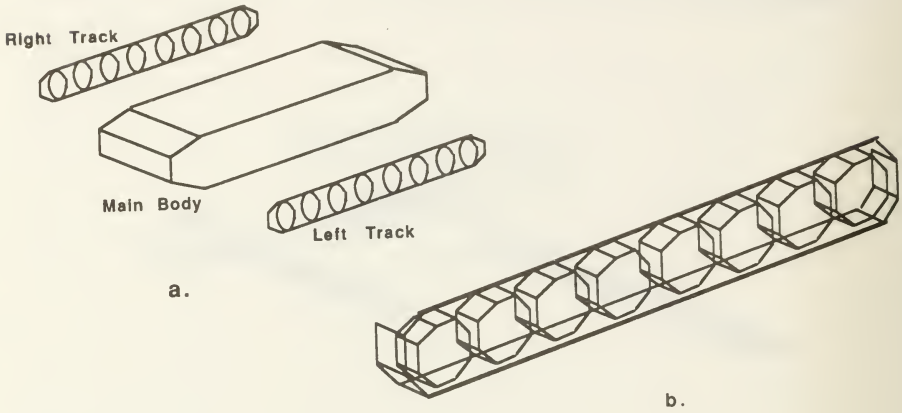


Figure 3-10. Tank Tracks

caused by the inside polygons of the left rollers being drawn over the right side of the main body (Figure 3-11 (a)). This distortion is hidden by using a color for the right side of the tank that blends with the color used for the track (Figure 3-11 (b)). Since lighting model calculations are performed only once in the simulation, this method is felt to be adequate.

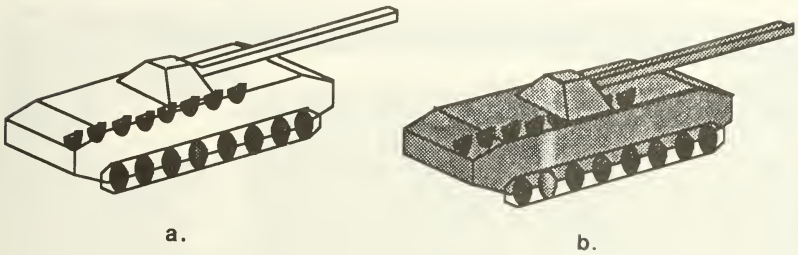


Figure 3-11. Tank Track Special Drawing Technique

All of the mentioned hidden surface drawing techniques are used to create a three-dimensional light shaded tank object (Figure 3-12). The use of various hidden surface techniques allow the tank to be viewed from any horizontal or vertical aspect without any distortion.

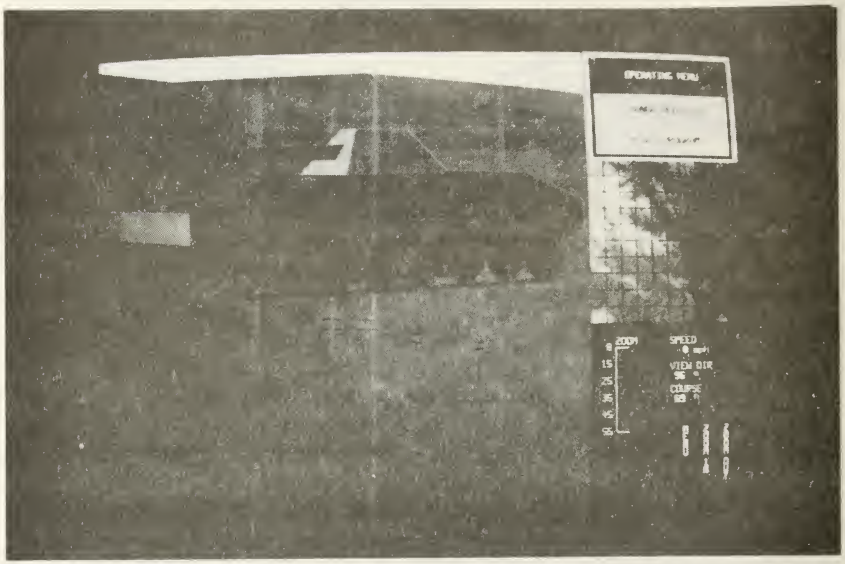


Figure 3-12. Tank Full Profile

2. Jeep

The jeep is a graphical object, similar to the tank, consisting of six parts: cabin inside, cabin outside, main body, tires, front headlights and rear taillights (Figure 3-13). These parts are drawn in an order to create a three-dimensional light shaded jeep. The techniques used to integrate the parts into a non-distorted object are discussed below.

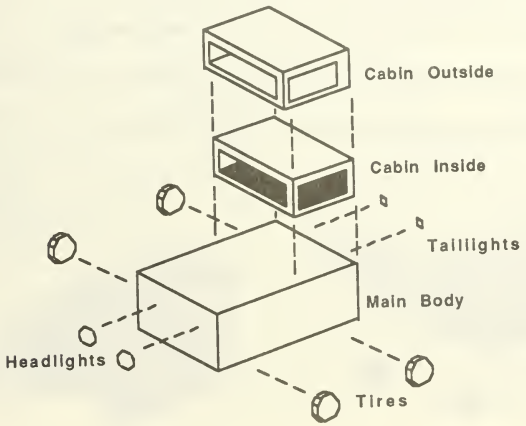
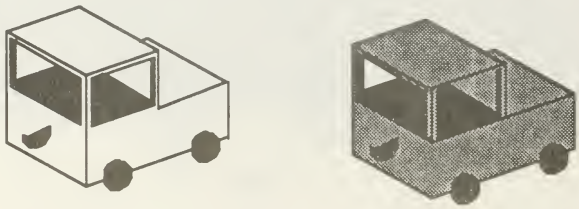


Figure 3-13. Jeep Parts

The jeep outer shell is built by first drawing the lower main body followed by the cabin. This drawing order causes the upper cabin portion of the jeep to paint over the lower body of the jeep when viewed from above. The tires and lights required a special drawing order when constructing the jeep's main lower body. A tire is a three-dimensional black octagonal object that is translated, rotated and drawn over the right or left side of the jeep's main body. The three-dimensional body, when drawn in the order of right side, right tires, left side then left tires, created the same distortion when viewing the jeep from the right side as did the tracks of the tank (Figure 3-14 (a)). To hide this distortion, the color for the right side of the jeep was made dark enough to blend in with the black tires (Figure 3-14 (b)). The headlights are drawn after the front part of the jeep to allow them to be seen from only a forward view. Each headlight is an eight sided white filled polygon that is translated and rotated into a position on top of the front part of the jeep. The same procedure is used for the taillights drawn on the back of the jeep.



a. b.

Figure 3-14. Jeep Tire Drawing Order

The jeep cabin had to be designed to allow views from both inside and outside the cabin. This problem was solved by breaking the jeep cabin into two separate parts. The inside part of the cabin is constructed of black polygons, all drawn using backface removal, to only allow them to be seen from the inside looking out (Figure 3-15 (a)). The outside of the cabin is then drawn at slightly larger dimensions so that it paints over the cabin inside. The result is a cabin that is not distorted when viewed from either the outside or inside (Figure 3-15(b)). All of these techniques create a three-dimensional light shaded jeep object, that can be viewed from another vehicle or a missile (Figure 3-16).

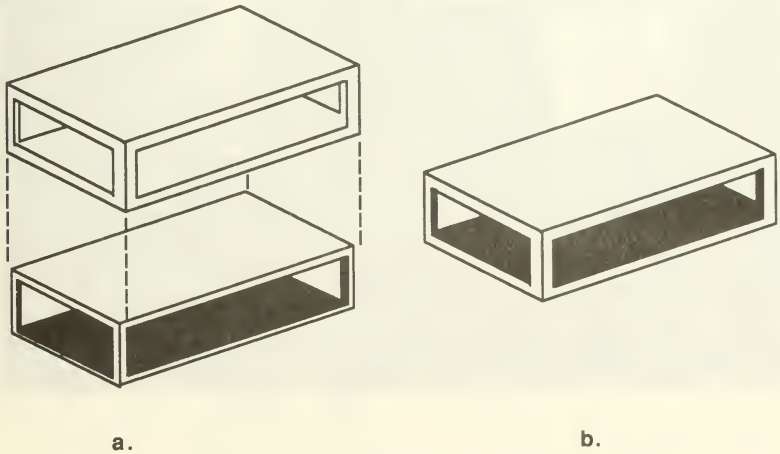


Figure 3-15. Jeep Cabin Drawing Order

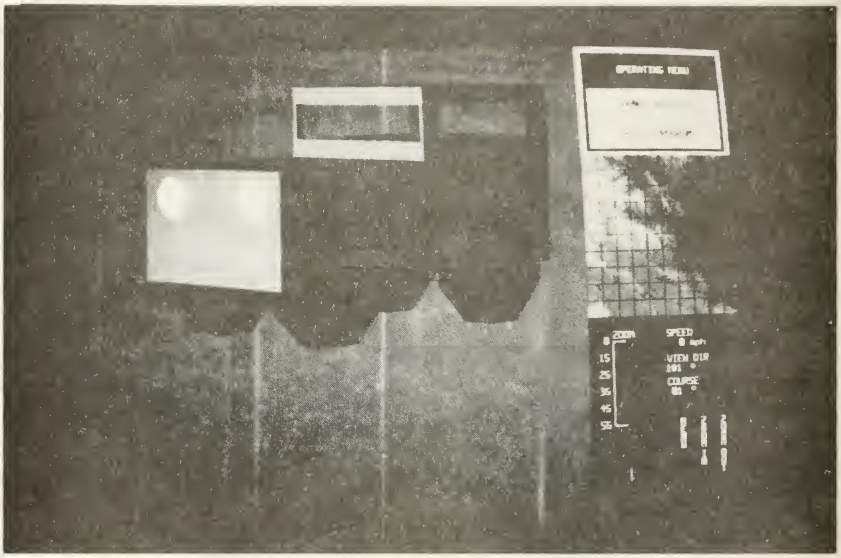


Figure 3-16. Jeep Full Profile

3. Truck

The truck is the most complicated three-dimensional object drawn in this study, consisting of seven parts: engine, cabin inside and outside, trailer, headlights, taillights and tires (Figure 3-17). The drawing of all these parts is ordered.

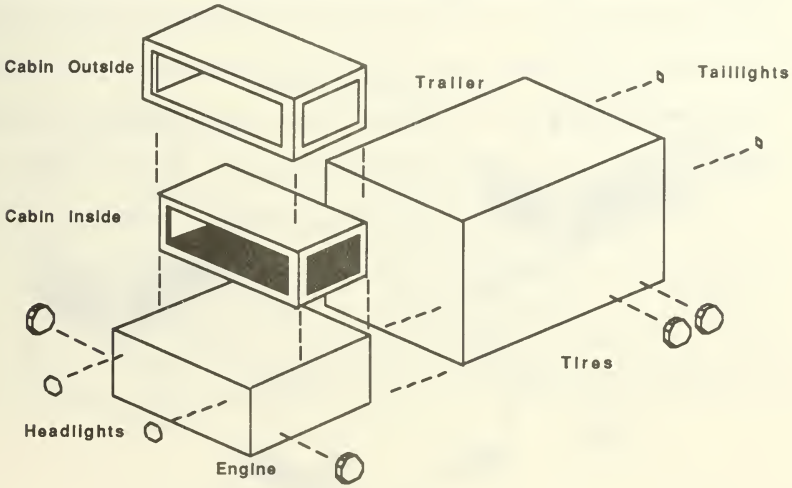


Figure 3-17. Truck Parts

The drawing order for the engine, cabin and trailer has to be done in such a manner as to display an undistorted view from both the front and top of the truck. This was achieved by first drawing the truck's trailer front, then the truck's engine and cabin (Figure 3-18). The cabin and engine parts paint over the truck's trailer when viewed from a front aspect, and the cabin paints over the engine when viewed from above. The lights and cabin were drawn in exactly the same manner discussed above for the jeep.

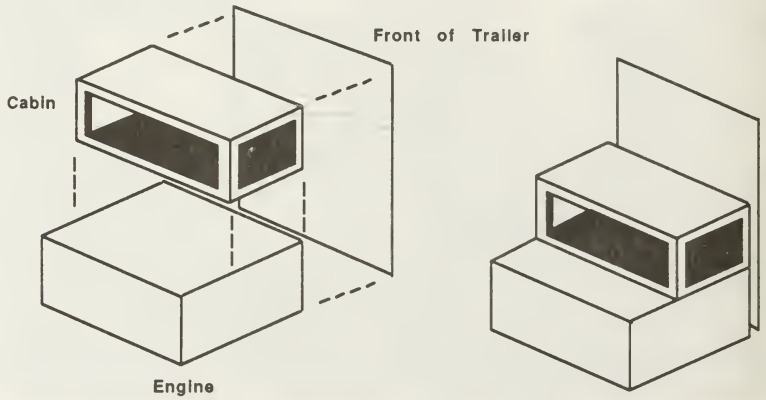


Figure 3-18. Engine, Cabin and Trailer Drawing Order

The trailer and tire parts required some additional drawing techniques not previously discussed. There are six tires used in drawing the truck, two front tires and four rear tires. The tire is the same one used for the jeep. Each front tire is rotated, translated and drawn after its respective side of the engine is drawn. The rear tire sets are also drawn after their respective sides of the trailer (Figure 3-19).

To eliminate the distortion of the inside sections of the tire that show through the right side of the trailer and engine, the color of the right side of the truck was selected to blend with the tires. An additional drawing distortion also occurs due to the trailer being drawn after the cabin. The right wheels paint over the left side of the engine when the truck is viewed from the left side (Figure 3-20(a)). This distortion was eliminated by drawing polygons over the distorted view after the right rear tires were drawn (Figure 3-20 (b)). All of these techniques create a three-dimensional light shaded truck object that can be viewed from another vehicle or a missile (Figure 3-21).

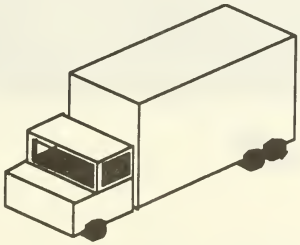
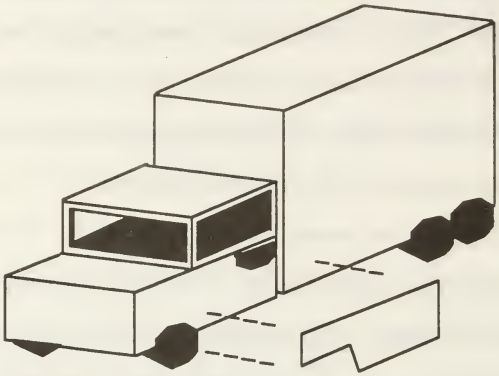
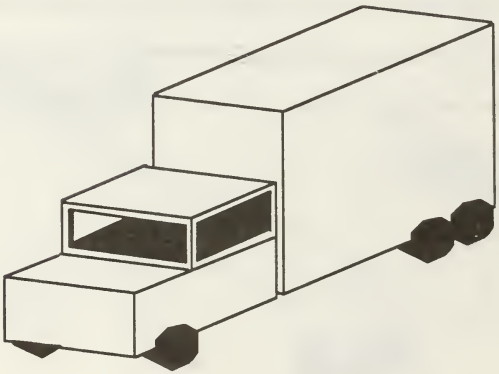


Figure 3-19. Truck Tire Drawing Order



a.



b.

Figure 3-20. Truck Tire Special Drawing Order

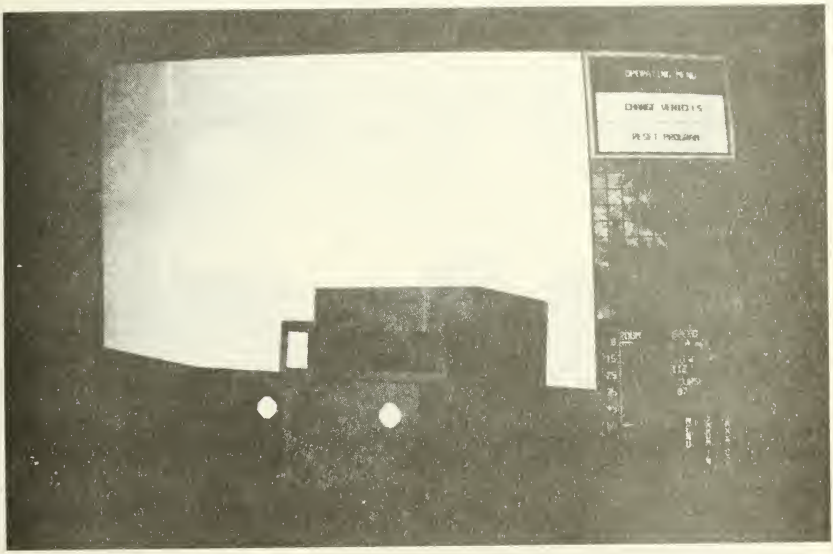


Figure 3-21. Truck Full Profile

4. Missile

The missile is a four-sided three-dimensional rectangular volume with a nose cap and smoke trail (Figure 3-22). The missile is a simple object due to the speed at which it travels in the viewing volume. At speeds of greater than two hundred knots, the missile usually appears as just a blur on the screen. No special drawing techniques were implemented to create the missile object. The main body of the missile is drawn first, followed by the nose cap and smoke trail. A full profile view of the missile, in a static position is shown in Figure 3-23.

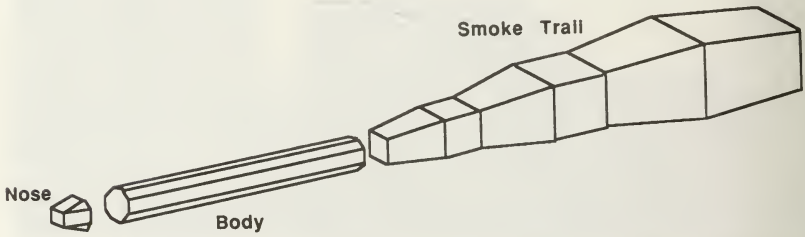


Figure 3-22. Missile Parts

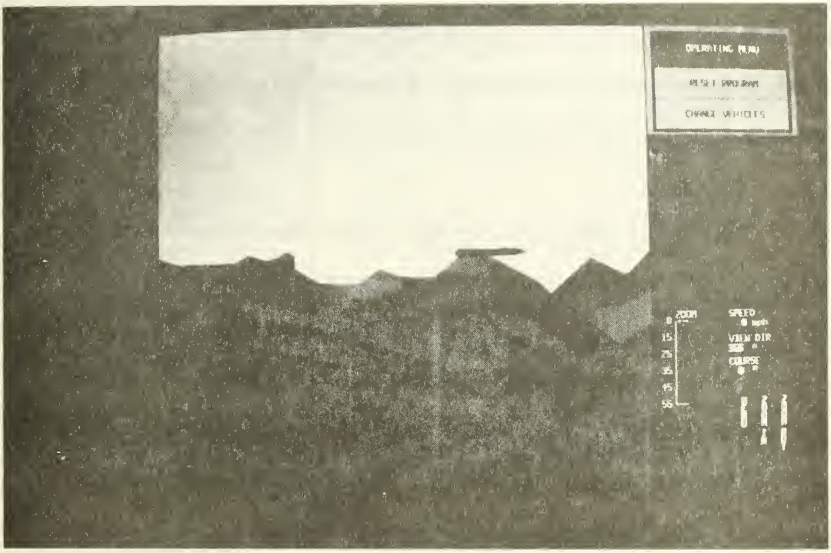


Figure 3-23. Missile Full Profile

D. TARGET ANIMATION

Target objects are built during program initialization. After the objects have been constructed, they are animated to fit in the proper orientation on the terrain. For example, a vehicle object's course and speed are used to redraw the vehicle each frame, at a distance proportional to the speed it would have travelled in the time it takes to refresh the screen. In addition to the speed and direction, a vehicle must be inclined to go up or down a hill, and tilted to go along a banked surface of a hill. The techniques used to animate and draw a target object on the terrain in real-time are performed in the display loop, after the targets' speed and direction parameters have been initialized. The algorithms used to perform these techniques are discussed in the following sections.

1. Initialization

The moving vehicle simulator uses two data structures to manage the correct display of vehicles in the scene. A linked list of vehicle definition data is created before the display loop begins, and is updated at each pass through the loop. Figure 3-24 depicts this structure. All of the information needed to transform a vehicle object to the correct position and orientation on the terrain is contained in the corresponding record in the linked list. Before the display loop begins, one graphical object is created for each type of vehicle. The drawing commands in this one object are then used to draw *every vehicle of that type,* instead of repeatedly adding drawing commands to existing objects, as was previously done. This technique alleviates the need for deleting and creating vehicle objects and improves the simulator frame update rate. The second data structure is used to solve the hidden surface removal problem. In order to use the painter's algorithm, the connection between grid squares and vehicles present in a grid square was

```
typedef struct vehicle {
```

short t	VEHICLE TYPE
float x	X TRANSLATION
float y	Y TRANSLATION
float z	Z TRANSLATION
short tilt	ROTATION ABOUT X AXIS
float ang	ROTATION ABOUT Y AXIS
short inc	ROTATION ABOUT Z AXIS
short gridx	X GRID VEHICLE IS DRAWN IN
short gridz	Z GRID VEHICLE IS DRAWN IN
float vel	VELOCITY IN METERS PER SECOND
float cse	COMPASS COURSE IN DEGREES
float dist	DISTANCE FROM DRIVEN VEHICLE
short hit	FLAG INDICATING MISSILE HIT
float sx	SCREEN COORDINATES WHERE ICON
float sy	SHOULD BE DRAWN ON CONTOUR MAP
short i	ARRAY INDEX FOR COMPATIBILTY WITH ORIGINAL FOG-M SIMULATOR
struct vehicle *next	POINTER TO NEXT RECORD IN THE LIST

```
} Vehicle
```

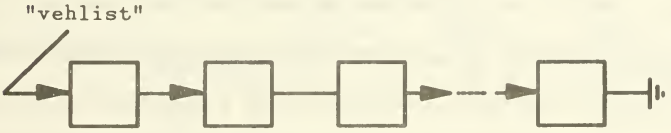


Figure 3-24 Vehicle Definition Data Linked List

maintained. A single two-dimensional array is created, with each element corresponding to a grid square, and with each element containing a list of pointers to records in the vehicle definition list. An element *vehgrid[Z][X]* in this array has pointers to definition data for only those vehicles that should be drawn immediately after grid square Z,X is drawn. These individual lists are maintained in sorted depth order, thus solving the hidden surface problem. An example is given in Figure 3-25. Four vehicles have been defined, creating the linked list *vehlist* of four vehicle definition data records. The vehicles are situated in grid squares Z,X and Z,X+2, in the arrangement shown. A reference point is required for depth sorting the vehicles before the *vehgrid* data structure can be created. This reference point is the position of the driven vehicle selected by the simulator operator, in this case vehicle 'B'. The linked list for *vehgrid[Z][X]* contains first a dummy node, and then pointers to vehicles 'A' and 'C' that appear in grid square Z,X. Each pointer is an address of a vehicle record in the *vehlist* data structure. Since vehicle 'A' is further from the driven vehicle than vehicle 'C' in this example, it appears first in the list. The linked list for *vehgrid[Z][X+2]* is similar, with vehicle 'D' further from vehicle 'B'. Drawing vehicles in a list correctly with respect to depth can be performed by traversing the linked list from its head, performing the appropriate transformations obtained from the definition data pointed to by the current node, then actually drawing the vehicle object. Objects are drawn with hidden surfaces correctly obscured if the list is maintained in a depth sorted order. A discussion of how vehicles are drawn can be found in the Chapter III section describing function *display_terrain()*. A discussion of how the *vehgrid* array is maintained is also found in Chapter III.

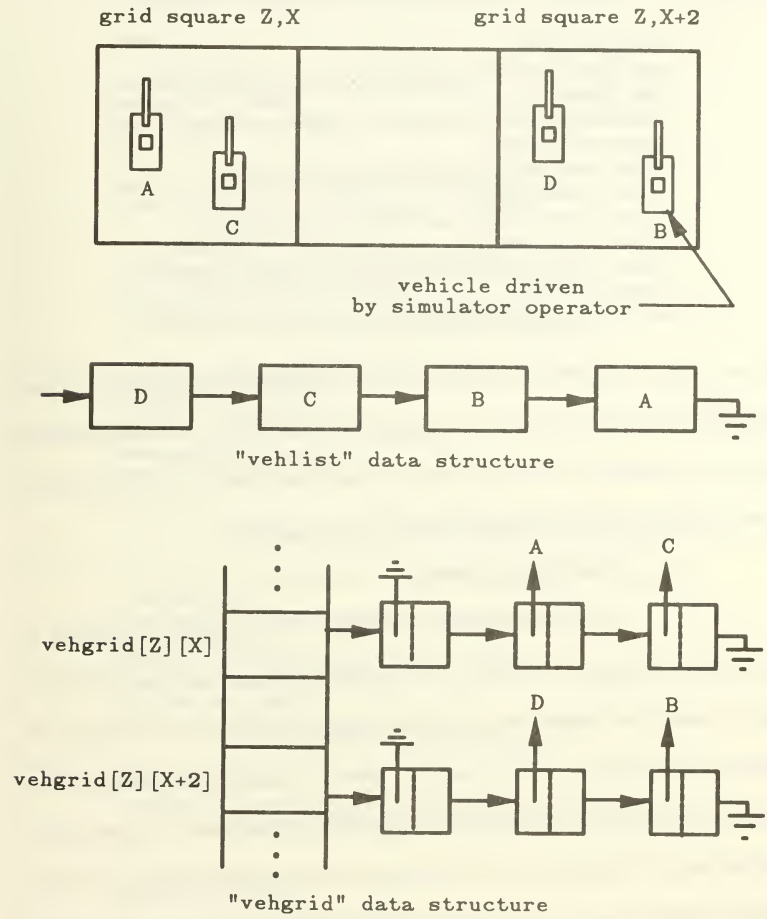


Figure 3-25 Vehicle Grid Array

2. Display Loop

All the functions used to draw the vehicles on the terrain are performed in the display loop. The display loop consists of the following six functions:

read_controls():
read operator controls to update driven vehicle parameters
view_bounds():
make field-of-view calculations for the driven vehicle
update_veh_pos():
update positions of all vehicles on the terrain map
update_vehicle_grid():
select the grid in which each vehicle should be drawn
update_look_pos():
update viewing orientation of the driven vehicle operator
display_terrain():
draw terrain map and target objects on the display screen

The actual display loop code consists of calls to the functions in the order shown. Each pass through the display loop represents a single frame of animation. All of the functions are optimized to produce a frame rate that simulates a real-time display. Each one of these functions are discussed in detail in the following paragraphs.

a. Read Operator Controls

Function *read_controls()* (Figure 3-26) allows the operator to interact with the program in real-time. The operator controls the driven vehicle by entering a course and speed, changing the viewing direction and changing the magnification of the scene viewed from the vehicle. All of these parameters are controlled by using either a mouse button or the dial box. There are eight knobs on the dial box that can be initialized and used as parameters for any graphics process. Five of these dials are used to control vehicle speed and course, driver tilt and line-of-sight look directions, and a color or black and white display (Figure 3-27).

```
#include "device.h"

read_controls(greys,active,lookang,tilt,lookdeg,fov)
Boolean *greys, *active;
float  *tilt,*lookang,*lookdeg;
short  *fov;
{
 extern Vehicle *driven;  /* pointer to the driven vehicle */
 float deltacsedeg,lastcsedeg= driven->cse;

/* if mouse button 2 Zoom in, if mouse button 1 Zoom out */
 if (getbutton(MOUSE2) && !(getbutton(MOUSE3)))
  *fov = (*fov < (150 + DELTAFOV)) ? 150 : *fov - DELTAFOV;
 if (getbutton(MOUSE1) && !(getbutton(MOUSE3)))
  *fov = (*fov > (550 - DELTAFOV)) ? 550 : *fov + DELTAFOV;

/* if DIAL5 is adjusted change the scene to Black and White using a
   greyscale color ramp */
 if (*greys != getvaluator(DIAL5)) {
  *greys = !*greys;
  setvaluator(DIAL5,*greys,0,1);
  colorramp(*greys,FALSE);
  }

  *tilt = DTOR * getvaluator(DIAL4) / TILTSENS; /* tilt head up and down */

/* Change the speed using DIAL2 If the speed is not zero read the course
   from DIAL0 */
 driven->vel = (float)(getvaluator(DIAL2) / SPEEDSENS);
 if( driven->vel != 0) {
  driven->cse = (float)getvaluator(DIAL0) / DIRSENS;
  deltacsedeg= driven->cse - lastcsedeg;
  if (driven->cse >= 360.0) {
   driven->cse -= 360.0; setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
            (int)(-360*DIRSENS), (int)(720*DIRSENS));
  }
  else if (driven->cse < 0.0) {
   driven->cse += 360.0; setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
            (int)(-360*DIRSENS), (int)(720*DIRSENS));
  }
 }
 else deltacsedeg = 0.0;
```

Figure 3-26. Reading Operator Controls

```
 if(deltacsedeg != 0) {
  *lookdeg= *lookdeg + deltacsedeg;
  if(*lookdeg >= 360.0) {
   *lookdeg -= 360.0; setvaluator(DIAL1,(int)(*lookdeg*DIRSENS),
                (int)(-360*DIRSENS), (int)(720*DIRSENS));
  }
  else if(*lookdeg < 0.0) {
   *lookdeg += 360.0; setvaluator(DIAL1,(int)(*lookdeg*DIRSENS),
                (int)(-360*DIRSENS), (int)(720*DIRSENS));
  }
  else setvaluator(DIAL1,(int)(*lookdeg*DIRSENS),(int)(-360*DIRSENS),
                                    (int)(720*DIRSENS));
 }
 else {
  setvaluator(DIAL0,(int)(lastcsedeg*DIRSENS),(int)(-360*DIRSENS),
                                    (int)(720*DIRSENS));
  *lookdeg = (float)getvaluator(DIAL1) / DIRSENS;
  if (*lookdeg >= 360.0) {
   *lookdeg -= 360.0; setvaluator(DIAL1,(int)(*lookdeg*DIRSENS),
                (int)(-360*DIRSENS), (int)(720*DIRSENS));
  }
  if (*lookdeg < 0.0) {
   *lookdeg += 360.0;
   setvaluator(DIAL1,(int)(*lookdeg*DIRSENS),(int)(-360*DIRSENS),
                                    (int)(720*DIRSENS));
  }
 }
 *lookang=(*lookdeg <= 90.0)? DTOR*(90.0- *lookdeg): DTOR*(450.0- *lookdeg);
 driven->ang = (driven->cse <= 90.0) ? DTOR*(90.0 - driven->cse)
                        : DTOR*(450.0- driven->cse);
}
```

Figure 3-26 (Continued). Reading Operator Controls

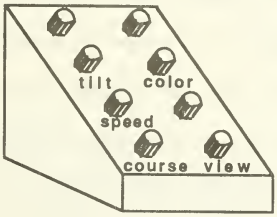


Figure 3-27. Dial Box

The use of the dial box requires initializing each dial to a parameter, with an upper and lower bounds and a sensitivity setting if desired

(setvaluator(DIAL#,parameter,lower bound,upper bound,sensitivity)).

The dial outputs a parameter by the function *getvaluator(DIAL#).*

Two mouse buttons are used to control the apparent magnification of the viewing volume. For every cycle of the display loop that the mouse button is depressed, the field-of-view is changed five degrees. Depending on the mouse button that is depressed, the field-of-view is either increased or decreased. The mouse button is sampled by the function *getbutton(MOUSE#).*

These controls allow the operator to drive a vehicle over any portion of the terrain, selecting any view out of the vehicle. These valuator inputs are constrained. The field-of-view is limited to fifty-five degrees to minimize the number of

polygons that have to be drawn. In addition, the driven vehicle's course cannot be changed if the vehicle is not moving. These constraints are discussed in the section *upvehpos()* and are implemented to provide more realistic moving vehicle dynamics.

b. Define the Viewing Boundary

Function *view_bounds()* (Figure 3-28) uses the field-of-view and view direction, entered by the operator in *read_controls()*, to compute the intersection of the right and left sides of the field-of-view with the terrain map boundary (Figure 3-29). These right (grx,grz) and left (glx,glz) coordinates are used by the *display_terrain()* function to draw only the polygons that are in the field-of-view.

```
view_bounds(lookang,fov,glx,glz,grx,grz)
Coord *glx,*glz,*grx,*grz;
float lookang;
short fov;
{
 extern Vehicle *driven;                    /* pointer to the driven vehicle */

 float halffov = DTOR*(float)(fov+50)/20.0; /* half of the field-of-view */
 float viewr  = lookang-halffov;            /* right half of field-of-view */
 float viewl  = lookang+halffov;            /* left half of field-of-view  */

/* left intersection points of the field-of-view and map boundary */
 intersection(driven->x,driven->z,viewl,&*glx,&*glz);

/* right intersection points of the field-of-view and map boundary */
 intersection(driven->x,driven->z,viewr,&*grx,&*grz);
}
```

Figure 3-28. Define the Viewing Boundaries

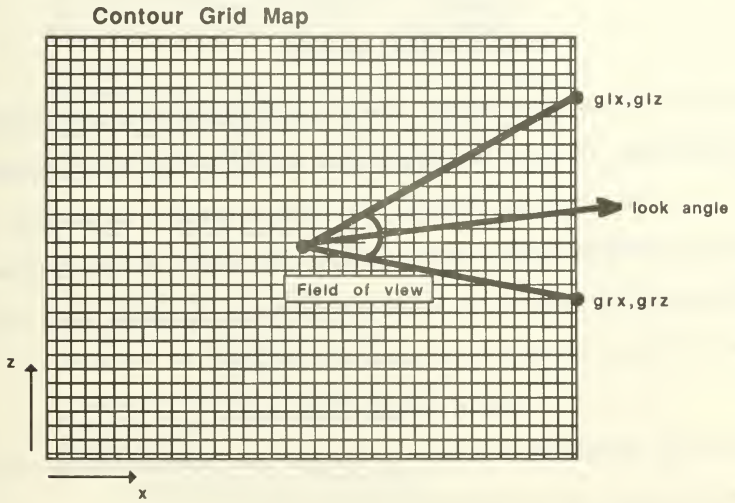


Figure 3-29. Viewbounds

c. Update the Vehicle Positions

Function *update_veh_pos()* (Figure 3-30) updates the position and orientation of all the vehicles in the vehicle list constructed during the initialization process. Each vehicle in the list has the following attributes:

x, y, z positions
velocity
course
type (TANK, TRUCK, JEEP)
distance from the driven vehicle
tilt angle
incline angle
hit by a missile (WRECK)

The values of these attributes maintain a vehicle's current location on the terrain map, its speed and course. The algorithms used to compute these attributes are discussed below.

(1) Speed and Direction. The vehicle's speed is implemented by translating it across the terrain at a rate proportional to its velocity. To compute the new translation position, the old position is added to the distance the vehicle travels in one frame. The distance the vehicle has travelled is first calculated.

$$distance = velocity * frame\ rate$$

Then the new positions are calculated by taking the sine and cosine of the course direction times the velocity and adding it to the old position.

$$x\_newpos = x\_oldpos + cos(course) * distance$$
$$z\_newpos = z\_oldpos - sin(course) * distance$$

The vehicle is then translated to the new coordinates (Figure 3-31). This translation also animates the direction that the vehicle travels, since the new translation position was derived using the vehicle's course.

```
update_veh_pos(elapsedsec)
float  elapsedsec; /* time to do one frame */
{
extern Vehicle *vehlist,*driven;
float  gnd_level(),sincos();
extern Boolean stall;

Vehicle *temp;
float  sine,cosine,distance;
float  ax,ay,az; /* incline point coordinates */
float  tx,ty,tz;  /* tilt point coordinates */

temp=vehlist;
while(temp!=NULL) { /* calculate new x,y,z position on the terrain */
 sine=sincos(temp->ang,&cosine);
 distance= temp->vel * elapsedsec; /* distance travelled in one frame */
 temp->x += cosine * distance;       /* new x coordinate */
 temp->z -= sine   * distance;       /* new z coordinate */

 /* calculate incline x and z coordinates */
 ax = temp->x + METERANDHALF *cosine;
 az = temp->z + METERANDHALF *sincos(((temp->cse - 90.0) *DTOR),&cosine);

 /* calculate tilt x and z coordinates */
 sine = sincos(temp->ang - HALFPI, &cosine);
 tx = temp->x + METERANDHALF * cosine;
 tz = temp->z - METERANDHALF * sine;

 /* compute tilt and incline y coordinates and add */
 /* the height to raise the vehicle above ground  */
 switch(temp->t) {
  case TANKS : temp->y = gnd_level(temp->x,-temp->z) +TANKGNDHT;
           ay = gnd_level(ax,-az) +TANKGNDHT;
           ty = gnd_level(tx,-tz) + TANKGNDHT;
           break;
  case TRUCKS: temp->y = gnd_level(temp->x,-temp->z) + TRUCKGNDHT;
           ay = gnd_level(ax,-az) + TRUCKGNDHT;
           ty = gnd_level(tx,-tz) + TRUCKGNDHT;
           break;
  case JEEPS : temp->y = gnd_level(temp->x,-temp->z) + JEEPGNDHT;
           ay = gnd_level(ax,-az) + JEEPGNDHT;
           ty = gnd_level(tx,-tz) + JEEPGNDHT;
           break;
 }
 temp->inc  =  (short)(asin((ay - temp->y)/METERANDHALF) * RTOD) * 10;
 temp->tilt = -(short)(asin((ty - temp->y)/METERANDHALF) * RTOD) * 10;
```

Figure 3-30. Update Vehicle Positions

```
 switch(temp->t) {  /* turn vehicle away from steep hills */
  case TANKS : if (temp->inc >80)    /* 8 degrees for a tank */
            slowturn(&temp->ang); /* turn vehicle 10 degrees*/
            break;
  case TRUCKS: if (temp->inc >100)  /* 10 degrees for a truck */
            slowturn(&temp->ang);/* turn vehicle 10 degrees*/
            break;
  case JEEPS : if (temp->inc >150)   /* 15 degrees for a jeep */
            slowturn(&temp->ang); /* turn vehicle 10 degrees*/
            break;
  }
  stall = FALSE;
  switch(driven->t) {       /* stall the vehicle if the hill is too steep */
   case TANKS : if(driven->inc > 80)  /* 8 degrees for a tank */
            stall = TRUE; break;
   case TRUCKS: if(driven->inc > 100) /* 10 degrees for a truck */
            stall = TRUE; break;
   case JEEPS : if(driven->inc > 150)  /* 15 degrees for a jeep */
            stall = TRUE; break;
  }
  /* if a vehicle reaches within 200 meters of map boundary turn it around */
  if((temp->x >(TENKM - TWOTENKM))||(temp->x < TWOTENKM)){
   slowturn(&temp->ang); /* turn vehicle 10 degrees*/
   if (temp == driven) /* if the driven vehicle stall it until it backs up */
   stall = TRUE;
  }
  else if((temp->z < -(TENKM - TWOTENKM)) || (temp->z > -TWOTENKM)){
   slowturn(&temp->ang); /* turn vehicle 10 degrees*/
   if (temp ==  driven)
   stall = TRUE;
  }
  /* if driven vehicle is stalled set the speed to zero until it backs up */
  if ((stall == TRUE)&&(driven-> vel >= 0.0)){
   setvaluator(DIAL2, 0 ,UPPERSPEEDBD,LOWERSPEEDBD);
   driven-> vel = 0.0;
  }
  temp=temp->next;  /* goto the next vehicle in the list */
 }
 temp=vehlist;
 while(temp!=NULL) { /* calculate distance from the driven vehicle */
  temp->dist=(float)hypot((long float)(driven->x - temp->x),
                    hypot((long float)(driven->y - temp->y),
                         (long float)(driven->z - temp->z)));
  temp=temp->next;
 }
}
```

Figure 3-30 (Continued). Update Vehicle Positions

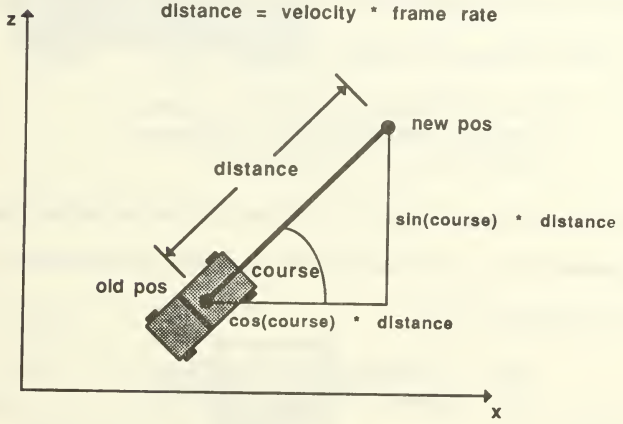


Figure 3-31. Vehicle Speed

All the vehicles head in their preset direction until they reach a map boundary or a steep hill. If a vehicle is not being driven and reaches a contour map boundary, it turns to the left, back into the center of the map by the function *slowturn()*. This function adds ten degrees to the vehicle's course each frame that it is within the two tenths of a kilometer of the grid map's boundary.

(2) Hill Traversal. All the vehicles encounter a hill at some time while traversing the terrain. To display the view of the vehicle object on a hill, the object must be oriented in both the incline and tilt directions. Two angles *inc* and *tilt* are computed by defining points one and a half meters away from the center of the vehicle's rotational axis. The x, y and z coordinates of this point are then calculated.

```
tx = temp->x + METERANDHALF * cos(course);
tz = temp->z - METERANDHALF * sin(course);
ty = gnd_level(tx,-tz) + TANKGNDHT;
```

The corresponding incline and tilt angles are computed by taking the difference (deltay) between the rotational axis y position and the points (ty or ay) y position and then calling the arcsine function (Figure 3-32).

```
inc = arcsin(deltay/METERANDHALF);
tilt = - arcsin(deltay/METERANDHALF);
```

Some of the terrain is too steep for a vehicle to traverse. The vehicle should either stall or turn to a less steep direction when attempting to go up a steep hill. If a vehicle's incline angle reaches a steepness threshold, the vehicle is turned to the left ten degrees for each frame of motion. This is accomplished by a call to *slowturn()*, which adds ten degrees to the vehicle's course. As long as the vehicle's incline angle is greater than the threshold, it keeps turning to the left. The driven vehicle is not turned

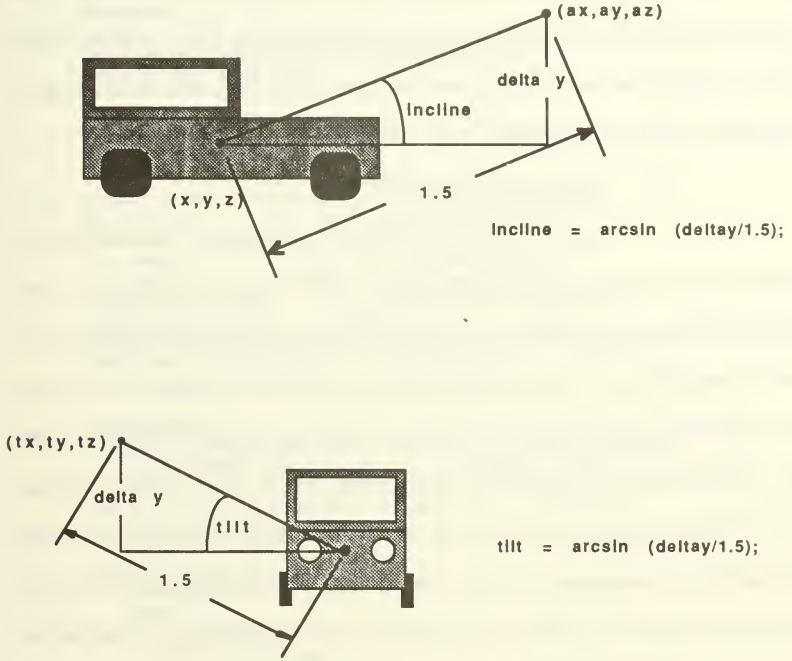


Figure 3-32. Incline and Tilt Computation

since the driver should be able to recognize and avoid steep hills. If the driven vehicle exceeds the incline threshold, it stalls and its speed is set to zero. The driven vehicle can then only be removed from the stalled condition by backing up and turning away from the steep hill, until the incline angle is less than the threshold.

Function *update_veh_pos()* is implemented using simple logic and mathematics. To reduce the time required to perform trigonometric functions, all the values for arcsine, sine and cosine are obtained by accessing an array consisting of 3600 entries. Once all the vehicle parameters have been computed, each vehicle must be assigned to a specific grid square.

d. Updating the Vehicle Grid Array

Function *update_vehicle_grid()* in the display loop manipulates the *vehgrid* array for correct vehicle object hidden surface removal (Figure 3-33). Recall that the *vehgrid* array has one element for each grid square, and that each element of the array is essentially a list of vehicles that should be drawn after the grid square is drawn. Two characteristics of this data structure are examined at each pass through the display loop. The pointers in a *vehgrid* array element linked list are sorted in order of distance from the driven vehicle to maintain correct vehicle drawing order within that grid square. The routine also determines which particular grid square a vehicle should be drawn after (which list it should be in) based on its proximity to a grid square edge, and the direction of the line-of-sight. This allows a vehicle to be drawn only once, regardless of its location on the terrain.

A vehicle situated near the center of a grid square is always drawn in the grid square it occupies. A vehicle near a grid square edge, however, can be drawn

```
#define WEST  0x8
#define EAST  0x4
#define SOUTH 0x2
#define NORTH 0x1

update_vehicle_grid(lookang)
float lookang;
{
 extern Gridnode *vehgrid[NUMZGRIDS][NUMXGRIDS];
 extern Vehicle  *vehlist, *driven;

 Vehicle *temp;
 short  ov,getoverlap(),quadrant,newxgrid,newzgrid,oldxgrid,oldzgrid;
 float  x, z;

 quadrant = (short)(lookang/HALFPI); /* quadrant driver is looking in.      */
 temp=vehlist;                  /* head of the definition data list.   */
 while(temp!=NULL) {            /* for each vehicle in the list do...  */
  x =  (temp->x);               /* get vehicle's current x position    */
  z = -(temp->z);               /*              ...and z position      */
  oldxgrid= temp->gridx;        /* get grid square indices vehicle was */
  oldzgrid= temp->gridz;        /*  last drawn in.                     */
  newxgrid=(short)(x/TENTHKM);  /* get grid square indices of vehicle's */
  newzgrid=(short)(z/TENTHKM);  /*  current position.                  */
  ov = getoverlap(temp->t,z,x,newzgrid,newxgrid);

  switch(quadrant) {
   case 0: if(ov & WEST)  newxgrid--; /* draw vehicle 1 grid square to west */
           if(ov & SOUTH) newzgrid--; /*          ...1 grid square to south */
           break;
   case 1: if(ov & EAST)  newxgrid++; /* ...etc.                           */
           if(ov & SOUTH) newzgrid--;
           break;
   case 2: if(ov & EAST)  newxgrid++;
           if(ov & NORTH) newzgrid++;
           break;
   case 3: if(ov & WEST)  newxgrid--;
           if(ov & NORTH) newzgrid++;
  }

  assign_grid(temp,oldxgrid,oldzgrid,newxgrid,newzgrid); /* move the node */
  temp->gridx=newxgrid;       /* update definition data record to reflect */
  temp->gridz=newzgrid;       /*  which grid square this vehicle is now   */
  temp=temp->next;            /*  drawn in                                */
 }
}
```

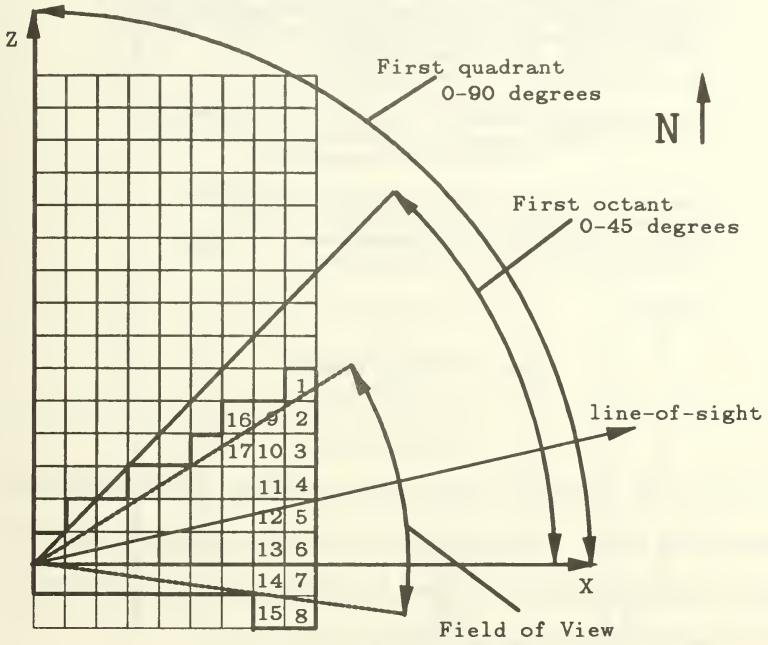
Figure 3-33. Determining Where to Draw a Vehicle

in the grid square it occupies or in an adjacent one, depending on the line-of-sight. *Update_vehicle_grid* relies on the algorithm used in function *display_terrain()* when determining in which list to place a vehicle (Figure 3-41 page 91). *Display_terrain()* draws terrain polygons and vehicle objects in a different order according to the direction of the line-of-sight. For purposes of *update_vehicle_grid()* calculations, the line-of-sight can be in one of four quadrants: the first quadrant extends from zero to one-half pi radians counterclockwise (ninety to zero compass degrees), the second quadrant extends from one-half pi to pi radians counterclockwise (zero to 270 compass degrees), and so on for the remaining two quadrants. Routine *display_terrain()* itself checks for presence of the line-of-sight in one of eight *octants*, with an octant being one-fourth pi radians (forty-five degrees). *Display_terrain()* drawing order is summarized in Table 3-1. An example with a line-of-sight in the first quadrant is given in Figure 3-34. Figure 3-43 (page 95) shows the eight octants applicable for *display_terrain()* calculations.

TABLE 3-1 DRAWING ORDER OF THE PAINTER'S ALGORITHM

Line-of-Sight angle	Quadrant	Drawing Order
$0 \leq LOS < 90$	0	North to South, East to West
$90 \leq LOS < 180$	1	North to South, West to East
$180 \leq LOS < 270$	2	South to North, West to East
$270 \leq LOS < 360$	3	South to North, East to West

A four bit "overlap" code is used to determine which edges a vehicle is near. Figure 3-35 shows routine *getoverlap()* that determines the overlap value. The appropriate bit of the overlap code is set if a vehicle is close enough to a grid square edge that it might overlap the adjacent square. A vehicle's proximity to an edge is obtained from the difference between its X and Z coordinates and those of the southwest corner of the grid square it occupies. Values above or below certain thresholds indicate



numbers indicate drawing order
north to south, east to west

Figure 3-34 First Quadrant Example Drawing Order

```
short getoverlap(type, z, x, zgrid, xgrid)
short type, zgrid, xgrid;
Coord x, z;
{
 float min, max, dx, dz;
 short zgrid, xgrid, ov=0;

 if(type==TANKS)  min=4.82; /* how close the center of the vehicle is to  */
 if(type==TRUCKS) min=4.54; /*   the west or south grid square edges when */
 if(type==JEEPS)  min=1.99; /*   the vehicle begins to overlap the       */
                            /*   the adjacent grid square.               */
 max = TENTHKM - min;      /* same as above, for north and east edges.   */

 dx = x - TENTHKM * (float)xgrid; /* dx = how close the vehicle actually  */
 dz = z - TENTHKM * (float)zgrid; /*   is to west edge of the grid square */
                            /* dz = how close to the south edge     */

 if(dx < min) ov |= WEST;   /* vehicle overlaps the western grid square   */
 if(dx > max) ov |= EAST;   /* ... eastern grid square                    */
 if(dz < min) ov |= SOUTH;  /* ... southern grid square                   */
 if(dz > max) ov |= NORTH;  /* ... northern grid square                   */

 return(ov);
}
```

Figure 3-35 Overlap Code Bits

some portion of the vehicle may overlap another grid square. These thresholds are different for each vehicle. The minimum threshold value for a tank is 4.82 meters for example, which corresponds to the diagonal distance from a tank center to one of its corners. Threshold values are illustrated in Figure 3-36.

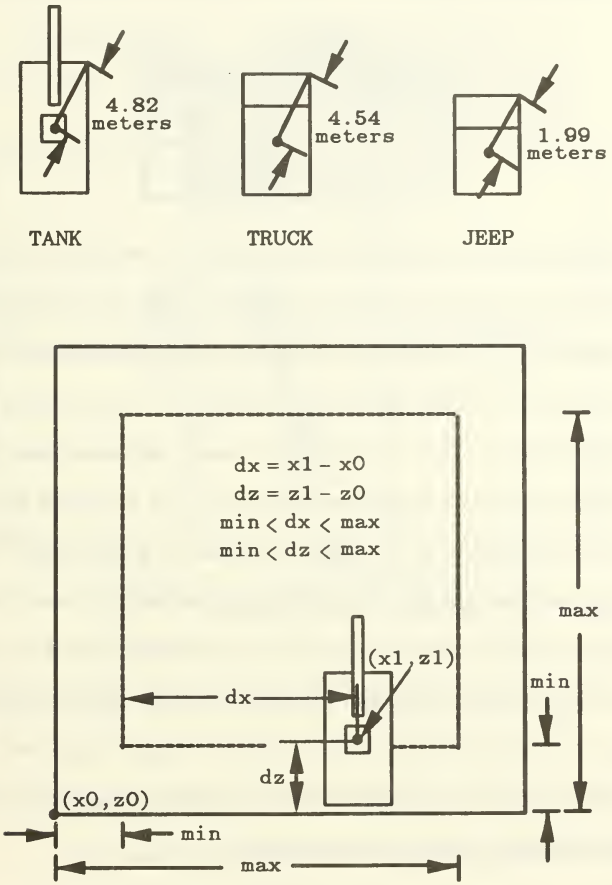


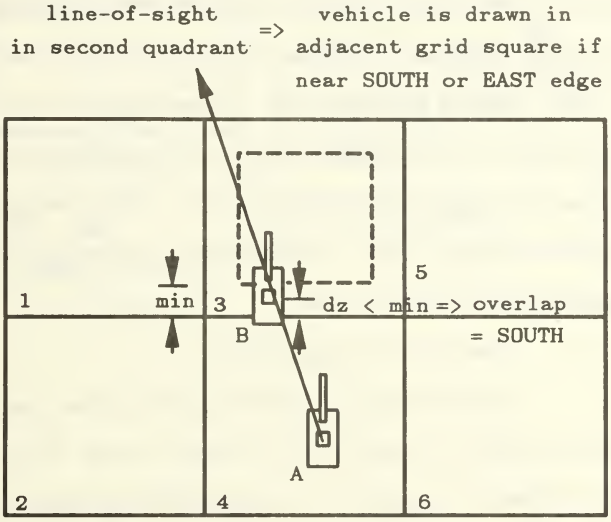
Figure 3-36 Proximity to a Grid Square Edge

Although a threshold is reached, a vehicle is drawn in the adjacent square only if it is near certain edges. It is the drawing order of the painter's algorithm that determines these grid square edges. They are shown in Table 3-2 for each quadrant.

TABLE 3-2

Quadrant	Grid Square Edge
0	South, West
1	South, East
2	North, East
3	North, West

In *update_vehicle_grid()*, the result of a logical AND of the overlap code bits and a bit representing one of the edges in Table 3-2 is used to determine the *vehgrid* list where a vehicle is found. Both bits must be set for a vehicle to be moved from one list to another. In Figure 3-37, the line-of-sight from the driven vehicle 'A' is in the second quadrant. With this line-of-sight, vehicles near a southern or eastern grid square edge are drawn after the adjacent grid square in that direction, instead of the grid square they occupy. This is the case for vehicle 'B' in Figure 3-37, near the southern edge. Since the painter's algorithm draws grid square four after grid square three, the part of the vehicle overlapping the southern grid square is overdrawn by grid square four if the vehicle is drawn after the grid square it occupies. To correctly draw the vehicle and both of the grid squares it overlaps, the vehicle must be drawn *after* grid square four. Routine *update_vehicle_grid()* and *getoverlap()* correctly determine the appropriate new *vehgrid* indices for this technique of hidden surface removal.



Vehicle 'B' is near SOUTH edge =>
draw it after grid square four

Figure 3-37 Drawing in an Adjacent Grid Square

The value obtained for the new Z and X grid indices is used by routine *assign_grid()* to move nodes in the *vehgrid* lists to their correct location. This routine consists of linked list management functions to search, remove and insert nodes in the *vehgrid* lists. Nodes are always placed in a list in depth sorted order.

An example clarifies the above discussion. Figure 3-38 shows the situation for two vehicles, with vehicle 'A' being driven. The upper arrangement shows the vehicles at the beginning of one pass through the display loop. Vehicle 'B' is located near enough to the center of grid square Z,X+1 that it does not overlap any adjacent squares. The lower arrangement occurs after vehicle positions have been updated. Vehicle 'B' is still located in grid square Z,X+1 but now overlaps the western edge. Since the line-of-sight is in the first quadrant, vehicle 'B' is drawn after grid square Z,X. The node for vehicle 'B' is moved from the *vehgrid[Z][X+1]* list to the *vehgrid[Z][X]* list based on an overlap code of "WEST". Again, insertion into the new list is done with the key being distance from the driven vehicle.

e. Updating the Viewing Orientation

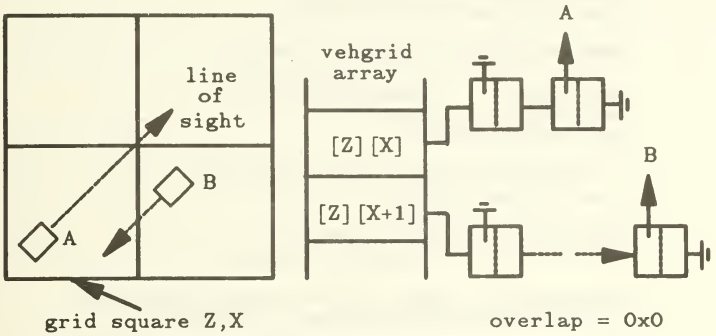
Function *update_look_pos()* (Figure 3-39) computes the reference point coordinates (px,py,pz) towards which the viewer is looking. The px and pz coordinates are calculated by computing the look distance in the line of sight, then taking the sine and cosine of the lookangle and adding the result to the viewer's x and z positions (Figure 3-40).

*px = driven->x + cosine(lookang) * lookdist*
*pz = driven->z + sine (lookang) * lookdist*

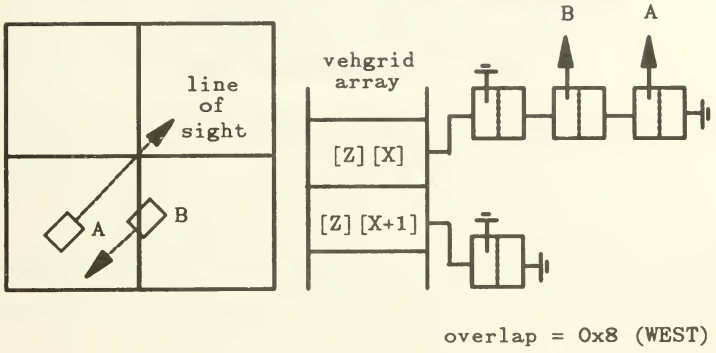
The distance the viewer is looking in the line-of-sight *(lookdist)* is calculated by taking a deltay value and dividing it by the tangent of the tilt angle.

lookdist = deltay/tan(tilt)

The *lookdist* is then normalized to a value less than the maximum look distance threshold. The py coordinate is calculated based on the viewer's tilt angle. If the tilt



(a)



(b)

Figure 3-38 Update Vehicle Grid Example

```
update_look_pos(lookang,tilt,px,py,pz)

float   tilt,lookang;
Coord  *px,*py,*pz;  /* reference coordinates */
{
extern Vehicle *driven;  /* pointer to driven vehicle */

float   lookdist;           /* distance ahead of viewer */
float   deltay;             /* height of eye */
float   sine,cosine,sincos();

   /* compute distance ahead of vehicle we are viewing */
deltay = driven->y - MIDELEV;

if (tangent(tilt) == 0.0)  lookdist = deltay / 0.01;
else                       lookdist = deltay / tangent(tilt);

if (lookdist < 0.0)        lookdist = -lookdist;  /*take care of tangent sign */
if (lookdist > MAXLOOKDIST)  lookdist =  MAXLOOKDIST;

   /* compute reference coordinate where viewer is looking */
sine=sincos(lookang,&cosine);  /* get sin & cos out of the array */
*px = driven->x + cosine * lookdist;
*pz = driven->z - sine   * lookdist;

if(tilt > 0.0) *py = 2.0 * driven->y;
else           *py = 0.0;
}
```

Figure 3-39. Update the Look Position

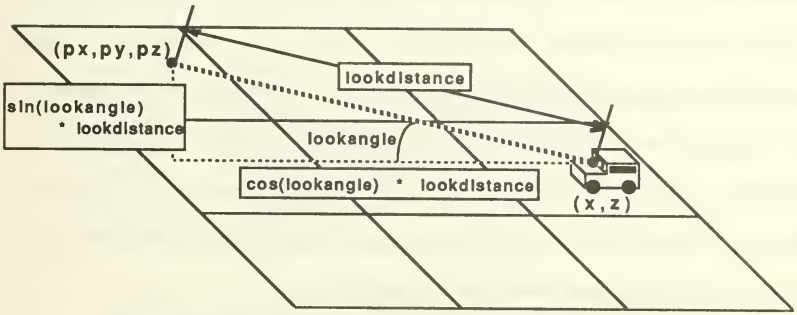


Figure 3-40. Calculating the Look Position

angle is above the horizon, py is set to look slightly above ground level. If the tilt angle is zero, the viewer is looking in the direction of the horizon and py is set to zero. The reference point is required to define the line of sight for the function *lookat(driven->x,driven->y,driven->z,px,py,pz,twist).* This function specifies the viewpoint (driven->x,y,z) and the reference point (px,py,pz) along the line of sight. The twist angle rotates the line of sight about the z axis. Once a viewing reference point is defined, the terrain and vehicles can be drawn.

f. Displaying the Terrain Map and Vehicles

Function *display_terrain()* draws the terrain map and vehicles in the viewing volume (Figure 3-41). The viewing volume line of sight, orientation and size are defined by the viewing transformations *lookat()* and *perspective().* The *perspective(fov,aspect,near far)* viewing transformation uses the field-of-view and near and far clipping planes to define the viewing volume drawn (Figure 3-42). The aspect represents how far the viewer sees in the x direction as compared to the y. For example an aspect of 3 means the viewer sees three times as far in the x direction as in y. The *lookat()* viewing transformation was explained previously.

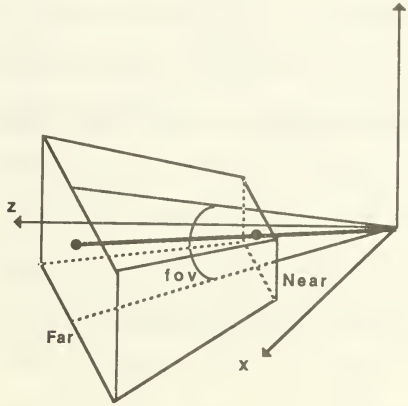
The screen is outlined in black with the sky colored blue and a ground plane of green. Only a 768 by 768 pixel square is used to display the scene. The rest of the screen is used to provide the operator with the driven vehicle parameters. After the scene has been initialized, the terrain map and vehicles are drawn. The details of how this is implemented are discussed below.

```
display_terrain(lookang,glx,glz,grx,grz, px, py, pz, fov,
                tank,jeep,truck,junk,intank, missile,networking)


Coord     px, py, pz;         /* viewing reference point */
Coord     glx, glz, grx, grz; /* intersect points with map boundaries */
float     lookang;            /* viewing angle */
int       fov;                /* field-of-view */
Object    tank,jeep,truck, missile,junk, intank; /* display objects */
Boolean   networking;



{

extern Coord     gndplane[4][3];
extern Vehicle   *driven;              /* driven vehicle pointer */
extern long      gndplane_color;       /* color for ground plane */
extern float     gridcoord[NUMZGRIDS][NUMXGRIDS][2][3][3];
extern float     height_of_eye[NUMVEHTYPES];
extern long      gridcolor[NUMZGRIDS][NUMXGRIDS];  /*color for grid
                                                     squares */
extern Gridnode  *vehgrid[NUMZGRIDS][NUMXGRIDS];
extern Missile   *msldata;


    Gridnode  *temp;  /* temporary pointer to vehicles */
    float   startx, startz, stopx, stopz; /*start and stop coord for scan lines */
    short   xgrid, zgrid;           /*x and z scan line grid indices */
    short   rotdir;                 /* rotation degrees */
    float   halffov = (DTOR * (float)(fov)/20.0);
    float   viewr  = lookang - halffov;
    float   viewl  = lookang + halffov;
    float   right, left;            /* right and left angles of fov */
    float   threshold;              /* max diagonal look distance */
    float   x =  (driven->x)/100.0; /* driven vehicle x grid index */
    float   z = -(driven->z)/100.0; /* driven vehicle z grid index */
    float   deltax, deltaz;
    float   tangent(); /* array of tangent values */
```

Figure 3-41. Display Terrain Initialization

```
viewport(0,767,0,767);          /* set the portion of the screen for the scene*/

pushmatrix();
color(SKYBLUE);                 /* color the sky blue */
clear();
ortho2(0.0,1023.0,0.0,767.0);   /* outline the screen in black */
color(BLACK);
recti(0,0,1023,767);
popmatrix();


pushmatrix();
 /* define the viewing transformations */
perspective(fov,1.0,0.0,MAXLOOKDIST);

lookat(driven->x,driven->y + height_of_eye[driven->t],driven->z,px,py,pz,0);

threshold = 20.0;  /* threshold for max number of grids drawn in LOS */

/* determine the direction of the line of sight  */
if (lookang < 0.0) lookang += TWOPI;


/* lay down a green ground plane to paint the terrain over */
color(gndplane_color);
polf(4, gndplane);


/*  ******************  Draw the Octant ******************  */

 /* code for a single Octant shown in Figure 3-44  below */

popmatrix();
```

Figure 3-41 (Continued). Display Terrain Initialization



perspective defines the field of view
and the near and far clipping planes

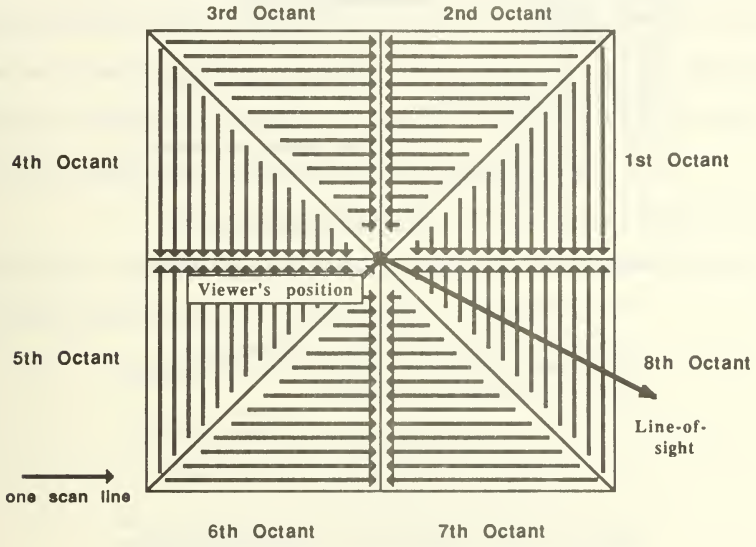
```
perspective(fov,twist,near,far);
    near  =  driven(x,y,z)
    far   =  reference point(px,py,pz)
    twist =  0;
```

Figure 3-42. Viewing Transformations

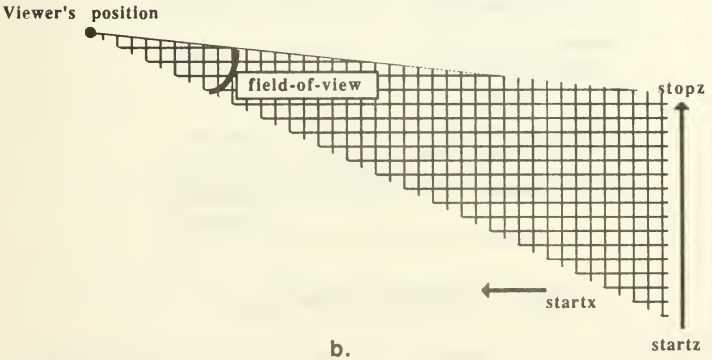
(1) Scene Display. The least number of grid squares is drawn by breaking the entire viewing circumference into eight parts or octants. The drawing order for each octant is based on drawing the grid squares, from the furthest to the nearest, using a scan line algorithm (Figure 3-43(a)). The viewer's position is in the center with the direction of the scan lines for each octant. As the field-of-view is changed, a different octant is selected. For example, if the field-of-view is in the eighth octant, the scan lines are defined by a startz and startx and then incrementing the startz until a stopz threshold is reached. This is one vertical scan line as shown in Figure 3-43(b). The next scan line is drawn by stepping the startx position towards the viewer, and repeating the process. Since each scan line is closer to the viewer and within the field-of-view, fewer grid squares are drawn each time the scan line is moved. For each grid square, the upper and lower halves of the terrain are drawn first, followed by any targets and the missile.

All of the octants are drawn using similar scan lines. Code for the eighth octant is shown in Figure 3-44. All the other octants use the identical algorithm. Note that the viewer's x and z positions are changed by one to draw the grid squares on adjacent sides of the viewer's grid square. This technique ensures that all the grid squares are drawn in the field-of-view.

After the entire scene is drawn, the vehicles in the viewer's grid square are drawn again. This ensures that any vehicles in adjacent grid squares, that may have been drawn after the viewer's grid square, are painted over by the viewing grid square's vehicles. Once we have descibed the technique for drawing the scene, the specifics on placing a vehicle on the terrain at the desired orientation must be discussed.



a.



b.

Figure 3-43. Octant Scan Lines

```
if (lookang > SEVEN_QTR_PI) {          /* The eighth Octant */
     startx = grx/100.0;               /*initialize start and stop  */
     startz = grz/100.0;
     stopz  = glz/100.0;

     if (startz <  0.0)   startz =  0.0;  /* ensure start and stop are on map */
     if (stopz  > 99.0)  stopz  = 99.0;

     startx =  x + threshold;          /* set the max number of grids drawn in
                                          the depth of field */
     if (startx > 99.0)  startx = 99.0;

     zgrid = (short)startz;            /* first z grid to be drawn scan line */

     while ((startx >= x) && (startz <= z)) { /* repeat until at the view pos*/

          xgrid =(short) startx;       /* set x  scan line */

          color(gridcolor[zgrid][xgrid]); /* color for the grid square */
          polf(3,&gridcoord[zgrid][xgrid][0][0][0]); /* draw the grid square */
          polf(3,&gridcoord[zgrid][xgrid][1][0][0]);

        /*******************Draw the vehicles ****************** */
                    /*  code shown in Fig. 3-45    */
       /* ******************* Draw the missile ***************** */
                    /*  code shown in Fig. 3-49    */

          zgrid += 1;    /* goto the next grid square on the scan line */

          if (zgrid > (short)stopz) {  /* completed a scan line */
           startx = startx - 1.0;      /* set the next x scan line */
           deltax =  startx - x;
           startz = z - (deltax * right);
           if (startz < 0.0)     startz = 0.0;

           if (lookang < AboveX_axis)
                stopz  = z - (deltax * left);
           else
                stopz  = z + (deltax * left);

           if (stopz > 99.0) stopz = 99.0;

           zgrid = (short)startz;  /*set the first z grid on the next scan line*/
      }
    }
  }
```

Figure 3-44. Displaying an Octant

(2) Vehicle Display. All the vehicles in a grid square are sorted in a linked list based on their distance from the viewer. After drawing the terrain of a grid square, the vehicles are drawn by traversing the linked list that is associated with the grid square's vehicle pointer. Because all the vehicles in the list are sorted by their depth in the viewing volume, proper drawing order is obtained (Figure 3-45).

Each vehicle must be drawn by first performing all the rotations, then translating it to a position within a grid square. Prior to any rotation, each vehicle object is drawn about the origin of a three-dimensional axis (Figure 3-46).

```
temp = vehgrid[zgrid][xgrid]->next; /*assign pointer to veh grid pointer*/
while ( temp != NULL) {             /*update every vehicle in the list */
  pushmatrix();
     rotdir = (short)(10.0 * RTOD * temp->vehptr->ang);
     translate(temp->vehptr->x,temp->vehptr->y,temp->vehptr->z);
     rotate(rotdir, 'Y');           /* rotate to course */
     rotate(temp->vehptr->inc, 'Z'); /* incline the vehicle */
     rotate(temp->vehptr->tilt,'X'); /* tilt the vehicle */

     switch (temp->vehptr->t){
      case TANKS  : if (temp->vehptr==driven) callobj(intank);
                    callobj(tank);        /* draw the tank */
                    break;
        case TRUCKS : callobj(truck); break; /*draw the truck  */
        case JEEPS  : callobj(jeep);  break; /*draw the jeep   */
        case WRECK  : callobj(junk);  break; /*draw the wreck */
        }
        popmatrix();
        temp = temp->next;    /* goto the next vehicle */
      }
```

Figure 3-45. Displaying the Vehicles

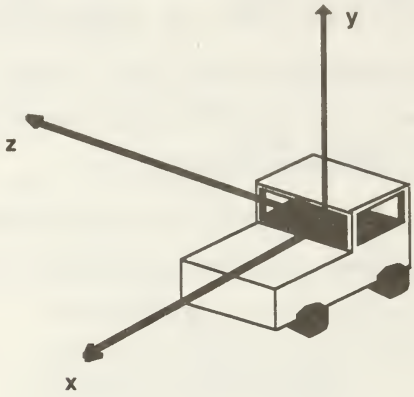


Figure 3-46. Vehicle Axis

To orient a vehicle object to the terrain, it is inclined by rotating it about the z axis and tilted by rotating it about the x axis.

```
rotate(temp->vehptr->inc, 'Z')
rotate(temp->vehptr->tilt, 'X')
```

In addition, the vehicle must be oriented to point in the direction it is heading. This is performed by rotating the vehicle object about the y axis. Note that all the rotations are performed while the vehicle object is still at the origin, then the translation is done (Figure 3-47). Only in this order will the vehicle be drawn at the

correct orientation at its new position on the terrain. These transformations are performed on the vehicle object in the following manner.

```
translate(temp->vehptr->x,temp->vehptr->y,temp->vehptr->z);
rotate(rotdir,'Y');
rotate(temp->vehptr->inc,'Z');
rotate(temp->vehptr->tilt,'X');
callobj(vehicle);
```

If the driven vehicle is a tank, another object called *intank* is drawn after the tank object. The *intank* object simulates the view the tank commander sees looking out of the tank. This view has slits bordered in black with a view of the top the gun barrel. A view from within a tank is shown in Chapter V. The *intank* object is drawn only when the driven vehicle is a tank. This is done to save time, since the polygons of the inside of the tank cannot be seen from the exterior of the tank.

(3) Missile Display. If the vehicle system is networking with another workstation that is running the missile simulator, the missile's course and position are passed via a communication link. When the grid square containing the missile is scanned, the missile object is rotated about the y axis to its course then translated to its position within the grid square (Figure 3-48).

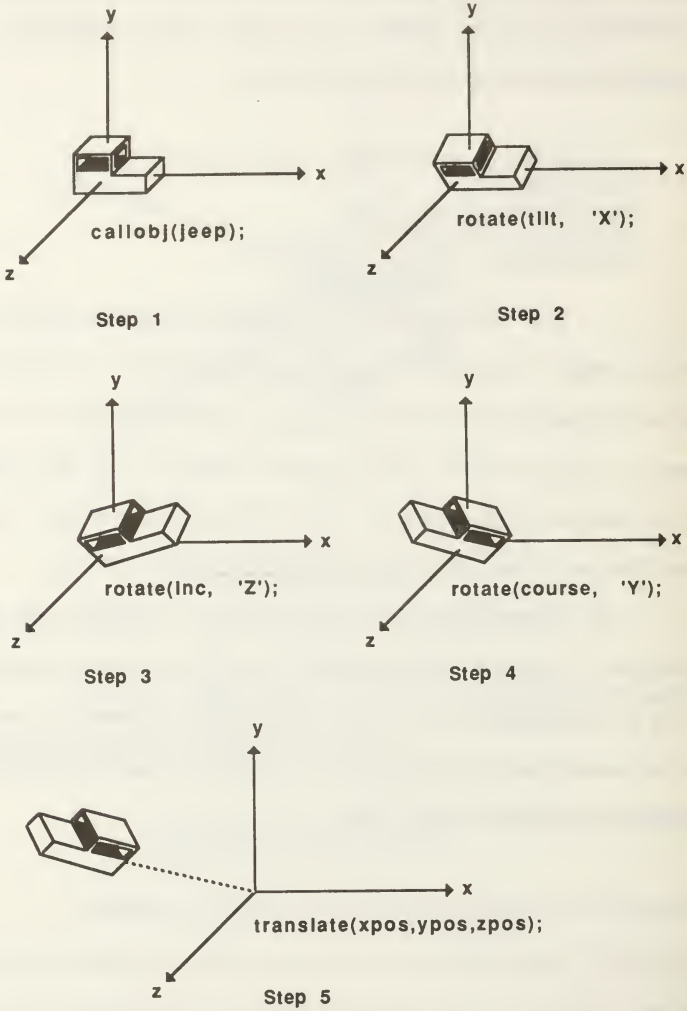


Figure 3-47. Vehicle Course

```
if(networking){      /* if communicating with missile simulator */
  if((msldata->gridx == xgrid)&&(msldata->gridz == zgrid)){
                       /*if missile is in scanned grid square*/
    pushmatrix();
      translate(msldata->x,msldata->y,msldata->z);
      rotate(msldata->cse, 'Y');  /* rotate to missile course */
      callobj(missile);           /* draw the missile */
    popmatrix();
  }
}
```

Figure 3-48. Displaying the Missile

(4) Destroyed Target Display. If a vehicle object has been hit by a missile, its vehicle type is changed to WRECK. The wreckage of a destroyed vehicle is displayed as a pile of twisted sides by drawing a *junk* object instead of the previous vehicle object when traversing the vehicle list in the function *display_terrain()* (Figure 3-49).

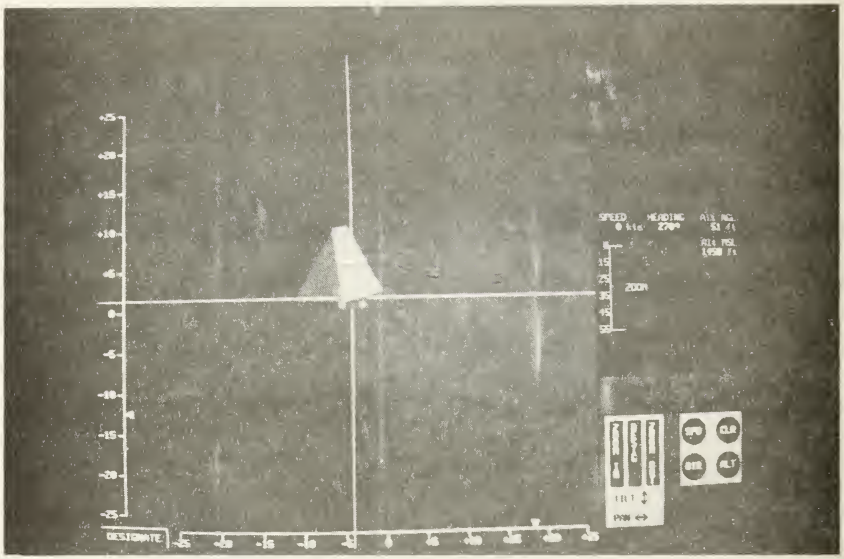


Figure 3-49. Destroyed Vehicle

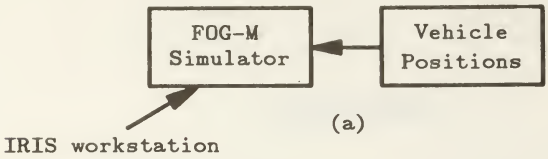
IV. NETWORKING

To provide as realistic a scene as possible to the FOG-M simulator operator, the targets he launches against must model real vehicles as closely as possible. This includes modeling the dynamic characteristics of a vehicle in motion as driven by a human operator. This realism was achieved by introducing a networking capability to the simulator, allowing a second workstation to provide interactive control over the movement of vehicles across the target area.

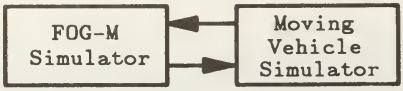
A. CAPABILITIES

The ability of the FOG-M simulator to receive information from an external source, as described in Ref [1] and depicted in Figure 4-1(a), was never implemented. The current study explored the use of 4.3BSD UNIX network capabilities to provide this intended feature. In addition, the goal of this study was to implement the broader capabilities of the system shown in Figure 4-1(b), to provide a single weapon console and a single target console in a two-node network of IRIS workstations. In this system, the interactions of each operator with the program running at his console are reflected in the opposite console. As the vehicle operator turns and accelerates his jeep, for example, the missile operator sees a turning jeep speed up as it moves across the terrain.

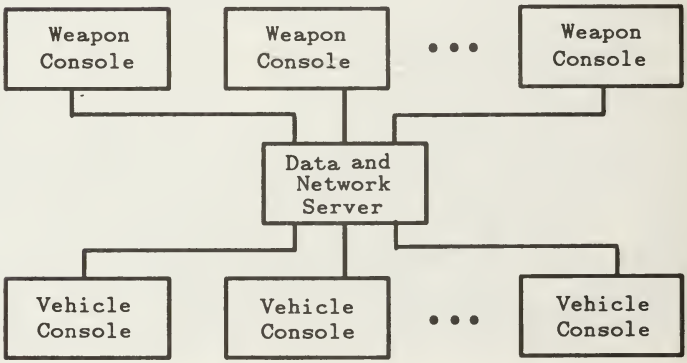
Each console in this system can act in stand-alone mode or can communicate with the other console. If networking is on, data and control information is exchanged between consoles to allow the vehicle operator to see the missile in-flight and to allow



(a)



(b)



(c)

Figure 4-1 Simulator Systems

the missile operator to view interactively controlled vehicles. The system provides for repeated weapons launch against multiple targets and the ability of either operator to exit from his simulator without affecting operation of the other console.

B. IMPLEMENTATION

This study was the first attempt at the Naval Postgraduate School to produce a network of interactive, real-time moving platform simulators. Normal (blocking) socket I/O was chosen as the network communications protocol for two reasons. Familiarity with this use of the 4.3BSD UNIX network session layer protocol was gained from the success of several smaller projects. Using blocking I/O results in the simulators operating synchronously, but it also aids in debugging the interaction of the two simulators. Since the normal result of reading an empty socket is for the reading process to block, this fact was used to isolate an improper sequence of socket I/O attempts. An excellent discussion of the UNIX socket mechanism is found in [5].

The vehicle console must use the current position and orientation of the missile to correctly render the missile image in the scene. Similar information is required by the missile console for displaying targets that have been established by the vehicle operator. Since this exchange of data is inherently duplex in nature, a pair of sockets was chosen as the mechanism to accomplish the transfer. The use of dedicated communication links in each direction and the guaranteed delivery of socket stream data ensures the availability and reliability of the necessary information.

The moving vehicle simulator was chosen to act as server to the client missile simulator, although the arrangement could be reversed. Operation of each simulator in a

networked mode requires the following steps in addition to those of stand-alone operation:

- initial set-up of the network
- initial data transfer
- display loop data transfer

Initial network set-up involves calls to system routines to establish the network data paths. Sockets are created to connect the workstations with a dedicated read path and a dedicated write path for both data and control information as indicated in Figure 4-2, for a total of four sockets opened by each simulator. Output to be sent to the other console is always sent via a simulator's "outxxx" socket, and all input is received via a simulator's "inxxx" socket. Failure to establish this configuration during initial network set-up results in fallback to stand-alone operation, with no further networking attempted. Data and control paths are different to maintain separation of function, and to allow the possible use of different transmission mechanisms for each path in future versions of the

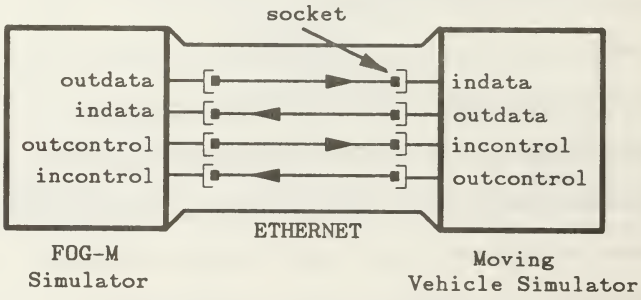


Figure 4-2 Network Connections

simulator. The synchronous operation of the current system might be avoided by using "out-of-band" or non-blocking mechanisms. This study did not explore these possibilities.

The number, type, and other relevant information about the vehicles defined by the moving vehicle simulator operator must be made known to the missile console before its operator can launch weapons against these targets (the sequence of steps used to define a vehicle are outlined in the User's Guide of Chapter V). After initial network set-up is completed, the FOG-M console waits to receive the vehicle definition data before permitting missile launch to occur. After the vehicle operator has defined the vehicles of his choice and has selected the vehicle he wishes to drive, routine *transfer_data()* passes this initialization data (Figure 4-3). The vehicle console then waits for missile launch. After receiving initialization information, the missile console allows a launch to take place. The vehicle console waits for the launch event before proceeding with the display loop to assure synchronization of the two programs at the time of missile launch. A handshaking takes place after initial data transfer and before the display loop begins to allow either console to exit from the simulation. This consists of each console sending its intent to continue or exit to the other console, with subsequent operation of the simulators based on the input received. If the missile console were to quit, for example, the vehicle console could continue to run in stand-alone mode.

Both programs enter their respective display loops to begin data exchange when the missile is launched. During the display loop, several key parameters can change. The position and course of the driven vehicle can change if its speed is non-zero. The

```
transfer_data(numveh)
short numveh[];
{
 extern Vehicle *vehlist;
 extern int    outdata;

 Vehicle *temp;
 char   outbuffer1[10], outbuffer2[80];
 int    i;

 for(i=0;i<NUMVEHTYPES;i++) {

  /* send the number of vehicles of this type */
  sprintf(outbuffer1,"%f",(float)numveh[i]);
  write(outdata,outbuffer1,10);

  temp=vehlist;
  while(temp!=NULL) {

   /* scan the definition data list for vehicles of this type */
   if(temp->t==i) {

    /* send position, velocity, and course data */
    sprintf(outbuffer2,"%f %f %f %f %f",
          temp->x,temp->y,temp->z,temp->vel,temp->ang );
    write(outdata,outbuffer2,80);
   }
   temp=temp->next;
  }
 }
}
```

Figure 4-3 Initial Data Transfer

position and course of the missile always changes during flight, up to the point of impact on a target. In addition, either operator can choose to exit his simulator during the display loop. Since all of these parameters directly affect the display, current values must be maintained. Figure 4-4 lists routine *network()* which handles display loop networking.

```
#include "veh.h"     /* defines CONTINUE=1.0 EXITING=0.0 */
#include "stdio.h"

network(networking,netwkestab,active,zoomed,scrn,menu,vehicon,
            windowsx,windowsy,mcnt,stat)
Boolean *networking,*netwkestab,active,zoomed;
Object  scrn[],menu[],vehicon[];
Coord   windowsx,windowsy;
short   *mcnt,*stat;
{
 extern Vehicle *driven;
 extern Missile *msldata;
 extern int     indata,outdata,incontrol,outcontrol;
        char    buffer[10],inbuffer2[60],outbuffer[80];
        float   flying,typehit,whichhit,mslcse,missile_console_status;
 /* HANDSHAKE */
 sprintf(buffer,"%f",(float)active);
 write(outcontrol,buffer,10);
 read(incontrol,buffer,10);
 sscanf(buffer,"%f",&missile_console_status);
 if(missile_console_status==EXITING) {  /* IF MISSILE CONSOLE HAS */
  *networking=FALSE;                    /* SECURED, DISCONTINUE   */
  exit_network();                       /* NETWORKING.            */
  *netwkestab=FALSE;
 }
 else {              /* WRITE VEHICLE POSITION, COURSE, IDENTITY */
  sprintf(outbuffer,"%f %f %f %f %f %f",driven->x,driven->y,driven->z,
              driven->cse,(float)(driven->i),(float)(driven->t) );
  write(outdata,outbuffer,80);
  read(incontrol,buffer,10);          /* CHECK FOR MISSILE HIT */
  sscanf(buffer,"%f",&flying);
  if((short)flying == TRUE) {     /* IF MISSILE IS STILL FLYING  */
   read(indata,inbuffer2,60);     /* READ ITS POSTION AND COURSE */
   sscanf(inbuffer2,"%f %f %f %f",&(msldata->x),&(msldata->y),&(msldata->z),&mslcse);
   msldata->cse = 10 * (short)(RTOD * mslcse);
   msldata->gridx=(short)( msldata->x / TENTHKM);
   msldata->gridz=(short)(-msldata->z / TENTHKM);
  }
  else {       /* FIND OUT WHAT IT HIT, AND HANDLE THE CASUALTY */
   read(indata,inbuffer2,60);
   sscanf(inbuffer2,"%f %f",&typehit,&whichhit);
   casualty(&networking,zoomed,scrn,menu,vehicon,windowsx,
         windowsy,typehit,whichhit,&*mcnt,&*stat);
  }
 }
}
```

Figure 4-4 Display Loop Data Transfer

The two simulators first handshake to ensure networking is still possible. Without this check, one simulator might attempt to read or write to a socket closed by a departing opposite console, aborting the simulator abruptly. With this check, when the opposite console exits, the continuing console is informed. In this case, it no longer attempts socket I/O, but can continue operation in stand-alone mode.

If the simulator can continue networking, the driven vehicle data is sent to the missile console. Even though more than one target can be present in the missile flight area, only driven vehicle data must be sent to the missile console. The driven vehicle is the only one that can be interactively controlled, while the others maintain course and speed. The missile console calculates a new position for each of the other vehicles based on the course and speed received from the initial transfer. After receiving the new vehicle information, the missile console sends a missile status flag indicating whether the missile is still flying or whether it has hit a target. In the former case, missile position and course data follows; in the latter case, the identity of the destroyed target is sent. If a vehicle has been hit, the vehicle console enters routine *casualty()*. If the identity of the destroyed vehicle matches that of the driven vehicle, an explosion is displayed. Otherwise the vehicle operator sees the burning wreckage of the destroyed vehicle if it is in the current field-of-view. In either case, the missile console then allows another weapon to be launched, and the vehicle console allows another vehicle to be selected for driving. At this point, either console can again elect to exit from the simulation without affecting the opposite console, as before.

C. LIMITATIONS

The implementation described above achieves the goal of providing out-the-window views from both missile and vehicle consoles in a network of two simulators, but it is limited in several respects. At present, the system allows the network connection of only one console of each simulator type, in a dedicated link arrangement. A more general distributed system such as that shown in Figure 4-1(c) would permit multiple workstations to "plug-in" to a central data and network server at will, entering and leaving the simulation at any time. Currently each simulator must perform both data processing and graphics output. A better solution would have the computation chore of updating vehicle and missile parameters done by the central server. Other consoles could then access this information to present their respective out-the-window views.

Due to the networking model chosen and the requirement of each console to update its own data, the simulators must run synchronously. A console does not proceed past its request for data from the other console (a socket read) until the information it needs becomes current. This assures the two displays remain identical with respect to vehicle and missile location and orientation in real-time. This "lock-step" executing nature of the simulators has the undesired affect of preventing the vehicle console operator from changing to a different vehicle to drive while the FOG-M missile is in-flight. Allowing this to happen would result in the missile "hanging" in mid flight until the new vehicle selection was made. When running in stand-alone mode, the vehicle console operator can reset the simulator or choose to drive a different vehicle at any time during the display loop.

While these limitations do exist, the system still provides a realistic though simple interactive simulation environment. The techniques used provide the basis for implementing the broader capabilities of the general distibuted simulation system of Figure 4-1(c).

V. MOVING VEHICLE SIMULATOR USER'S GUIDE

A. INTRODUCTION

The user-interface of the original FOG-M simulator has been more fully developed to present a user-friendly, easy-to-use system. The primary operator interaction with the system is through a series of menus providing all of the available user options. Help information appears on-screen with each of these menus. In addition, the use of icons, color and sound gives visual and audio feedback during most user actions.

Operation of the simulator consists of an initialization phase where the simulator configuration is established, and the actual simulation phase where vehicles travel across the terrain. In the first phase, the user must indicate whether the simulator is to be run in stand-alone mode or in conjunction with the FOG-M simulator. He must also define each vehicle that is to be present in the simulator, to include vehicle locations, course and speed. The simulation phase allows interactive control of one of the defined vehicles and displays the out-the-window view from this driven vehicle.

B. INITIALIZATION

After logging on to an IRIS workstation, typing

veh [connect_to]

at the UNIX prompt loads the moving vehicle simulator program and begins its execution. The executable module of the simulator must be present in the current

directory for it to be started in this manner. The command line argument *connect_to* is the network host name of an IRIS workstation to be connected to should the user decide to run the simulator in its networked mode. A default workstation is set if this argument is not entered.

The simulator makes use of two external files that must be available to continue normal program execution. Terrain elevation data is read from the file **dted.veh** found in the path **/work/DTED.** If this file cannot be found, the simulator displays a red warning screen and the message

TERRAIN ELEVATION DATA FILE NOT FOUND.
Do you wish to continue?
If continuing:
(1) the terrain will appear flat
(2) networking will be disabled
Enter 'q' to quit.
Enter 'c' to continue.
RESPONSE ==> ?

Continuing execution without terrain elevation data is possible, but in this case the entire 100 square kilometer area of terrain is drawn at the same (zero meters) elevation. This does not affect the moving vehicle simulator except with respect to networking. The terrain elevation database is not a shared resource: each simulator uses its own local copy of elevation data. Since the FOG-M simulator expects a non-zero value for a vehicle's Y (elevation) coordinate, translation of a vehicle to zero meters height in the FOG-M display draws the vehicle incorrectly. Networking is disabled to prevent this. All other features of the moving vehicle simulator remain present, but the view from the inside of a vehicle is of an uninteresting flat expanse of checkerboard colored ground.

If the user continues, the simulator proceeds to read file **polygon.data** also found in the path **/work/DTED.** This file of terrain polygon colors is created if it is not found. Colors are read (or written) with a brief countdown appearing on the screen during the process. At this point, the opening menu and first introductory screen appears describing the simulator and its features.

The current menu is always present in the upper right corner of the display. Instructions that apply to the current menu appear in the lower right corner of the display. Menu selections are made with the left mouse button by moving the cursor to the desired menu item, then depressing and releasing the button. Mouse movements are constrained to the menu area when menu selections are the only input possible. Menu items appear in a yellow color if the cursor is moved off the menu; they appear red when the cursor passes over an item, and they are highlighted in white when the selection is made. It is possible to abort a menu selection after depressing the left mouse button simply by moving the cursor off the selected item. Invalid selections cause the keyboard bell to ring several times to indicate the invalid choice. This can occur if a menu item is selected when that item is not currently available. Supplementary information describing the invalid selection appears in the menu instructions area of the screen should this occur.

The below listed six menus provide all of the available user options for controlling the simulator:

OPENING MENU
MAIN MENU
ADD VEHICLE MENU
DELETE VEHICLE MENU
SWITCH VEHICLES MENU
RUN MENU

1. Opening Menu

Menu choices available from the OPENING menu are:

NEXT PAGE
PREVIOUS PAGE
NETWORKING
QUIT PROGRAM

Three screens containing introductory textual information can be paged through from the OPENING menu using the *NEXT PAGE* and *PREVIOUS PAGE* menu items. If networking is desired, the *NETWORKING* menu item must be selected. The operator can also choose the *QUIT PROGRAM* item to exit the simulator. This item is available in most menus.

2. Main Menu

a. Options

The MAIN menu is entered by selecting *NEXT PAGE* from the third introductory opening screen. The MAIN menu appears along with a large two-dimensional contour map of terrain (Figure 5-1(a)). All initialization phase actions are carried out from the MAIN menu. MAIN menu item selections result in vehicles being defined or the actual simulation begun or exited, with choices made from the following items:

ADD VEHICLE
DELETE VEHICLE
DEFAULTS
RUN
ZOOM IN/OUT
QUIT PROGRAM

Selecting *ZOOM IN/OUT* with the large contour map present on the screen allows the user to view a small one-kilometer area of the map in larger scale.

This is done by moving the cursor to the desired map location and depressing the left mouse button. If a zoomed-in section of the map is displayed, selecting *ZOOM IN/OUT* returns the display to the large contour map. Figure 5-1(b) shows the zoomed-in contour map.

Selecting *RUN* completes the initialization phase and begins the simulation. RUN is discussed below in the section describing the display loop.

b. Defining Vehicles

Vehicles are defined by adding new vehicles or deleting previously defined vehicles. Vehicles can be added by selecting the *DEFAULTS* menu item or the *ADD VEHICLE* menu item. *DEFAULTS* places three vehicles of each type (TANKS, TRUCKS, JEEPS) on the map near the middle right area of the terrain, all traveling north at a speed of about twelve miles-per-hour. The user can select *RUN* to begin the simulation with only these vehicles, or can continue to add or delete vehicles as described below. Vehicles appear on the contour map as a small icon with an arrow pointing in the travel direction.

Selecting *DELETE VEHICLE* allows a user to remove a previously defined vehicle from the contour map. The cursor is changed to an 'X' shape, and must be positioned over the vehicle to be deleted. Depressing the left mouse button then removes the vehicle from the map.

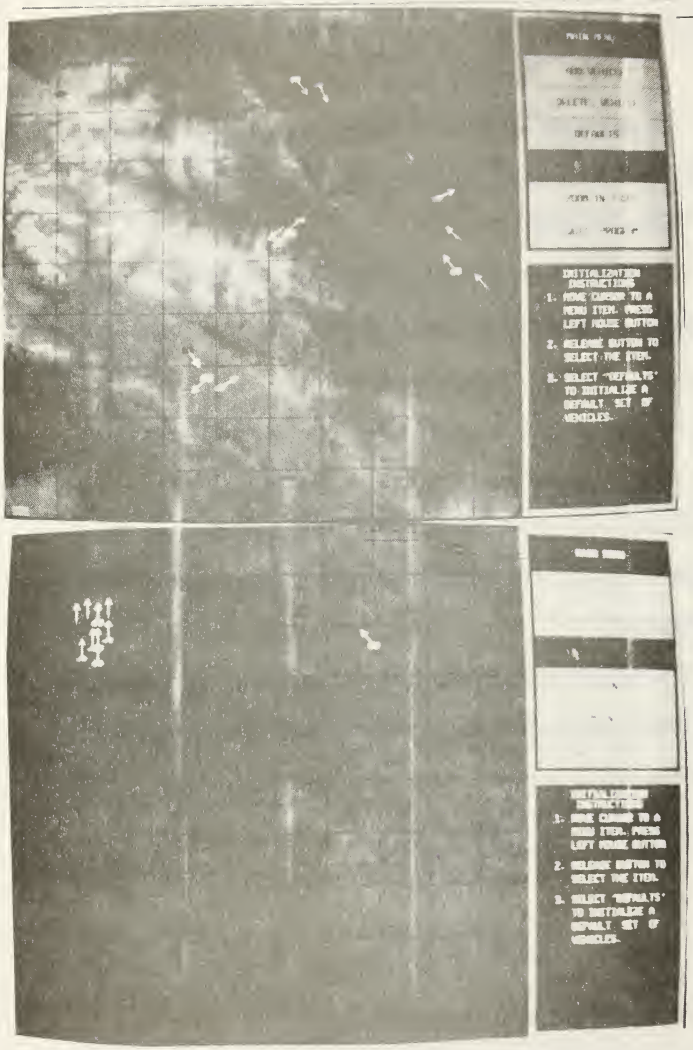


Figure 5-1. Contour Displays

The *ADD VEHICLE* menu choice displays the ADD menu with the following selections available:

ADD TANK
ADD TRUCK
ADD JEEP
RETURN TO MAIN

The first three items define a vehicle's type, and the last item returns the display to the MAIN menu. After selecting a vehicle type, the cursor can be moved from the menu area to a location on the contour map. The cursor changes to an icon depicting the selected vehicle type. A vehicle's location on the terrain is set by moving the cursor/icon to the desired location, then depressing the left mouse button. An icon image of the vehicle appears on the map at the specified location. The vehicle's course is set in a similar manner. Once the vehicle has been placed on the map, moving the cursor gives a "rubber-band" line from the icon to the current location of the cursor, indicating a possible course for the vehicle. Depressing the left mouse button sets the course to the direction of the rubber-band line. After a vehicle's position and course has been established, a speedometer appears in the menu area of the display to allow setting the vehicle's speed. The speedometer is a sliding rectangle contained in a rectangular box marked in miles-per-hour increments, with the current speed appearing below the speedometer. The cursor is automatically placed on the slider bar at an indicated speed of zero once the vehicle course is set, and can be moved left or right to change the speed. Depressing the left mouse button sets the speed to the value shown below the speedometer.

To define several vehicles of the same type the sequence described above

- move cursor/icon to desired map location
- move rubber-band line to desired course
- move cursor to desired speedometer speed

can be repeated without returning to the MAIN menu. Once all vehicles have been defined, selecting *RUN* from the MAIN menu displays the large contour map, icons for all defined vehicles, and the SWITCH VEHICLES menu.

3. Switch Vehicles Menu

To begin the simulation and provide the user with an out-the-window view from a vehicle, the vehicle to be driven must be selected from those previously defined. SWITCH VEHICLES menu options are:

ZOOM IN/OUT
QUIT PROGRAM

As described before, the *ZOOM IN/OUT* item allows a closer look at a small area of the contour map. A vehicle is selected for driving by moving the cursor over the vehicle's icon on the map, then depressing the left mouse button. The cursor changes shape to a crosshair as it moves out of the menu area. Selection of a vehicle on the map begins the display loop operation of the simulator. If networking, the vehicle simulator waits until missile launch occurs to enter the display loop. The options available for interactive control of vehicles is described in the next section.

C. DRIVING CONTROLS

After *RUN* is selected from the MAIN menu and the driven vehicle is selected, the display changes to the terrain and vehicles, with a view from inside the driven vehicle (Figure 5-2). The driving display is divided into five parts:

- The terrain viewed from the driven vehicle
- The vehicle control panel
- The navigational status panel
- A scaled contour map
- The operating menu bars

The techniques used to control the driven vehicle and displays are discussed below.

1. Driven Vehicle Controls

The driven vehicle's course and speed can be changed by using the dial box. The course and speed ranges are 0 to 360 degrees and -40 MPH to 60 MPH respectively. The sensitivity is set to provide a smooth transition of values throughout the range of each dial. The operator is given four means of viewing the settings that are entered.

- Digital displays for speed in MPH and course in degrees.
- Relative motion between driven vehicle and terrain
- An arrow on the small scale contour map for course
- Motion of the contour map arrow for speed.

While driving on the terrain, the driven vehicle can stall if either of the following conditions are satisfied.

- Vehicle enters within 200 meters of a terrain map boundary
- Vehicle exceeds a steepness threshold on a hill

The only way to remove the driven vehicle from a stalled condition is to back it up and change the course away from the condition that caused it to become stalled originally.

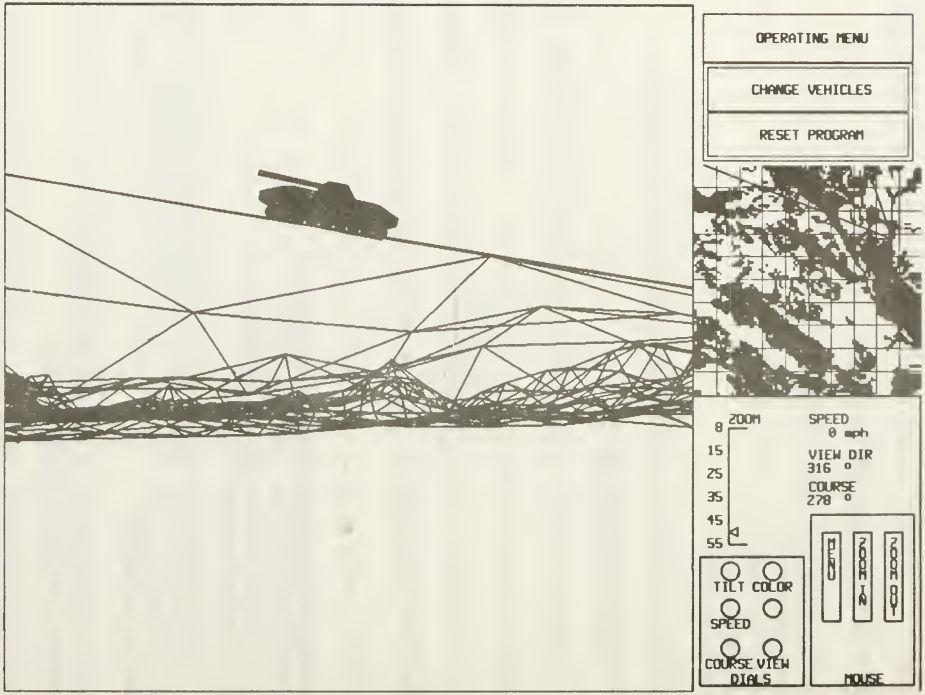


Figure 5-2. Driving Display

2. Driven Vehicle Views

When driving, the operator views the terrain from the inside of a vehicle. The view from within a vehicle can be changed by using either one of two dials. These two dials change the direction the operator is looking out of the vehicle and how far his head is tilted up or down. The direction out of the vehicle can be changed from 0 to 360 degrees relative to the vehicle's course and the tilt from -25 to +25 degrees with respect to the horizon. The view out of the jeep is unrestricted in all directions except for the posts that support the roof (Figure 5-3). The view out of the tank is limited to 82 degrees to simulate the restricted view of the tank commander (Figure 5-4). The view out of the truck is limited to 180 degrees because it has no back window. All the restricted views are displayed to the operator as black surfaces.

3. Menu Selections

During the driving display, the operator is given two menu choices at the upper right hand side of the screen. If not networking, the driver can switch to another vehicle by using the left mouse button and selecting *CHANGE VEHICLES*, or the number and placement of all the vehicles on the terrain can be erased by selecting *QUIT*. If the system is networking, the *CHANGE VEHICLES* selection is not available, and *QUIT* halts the program.

If the operator selects *CHANGE VEHICLES*, the display changes to the large contour map with the vehicle ICONS. The menu selections are now the same ones used during the initialization sequence just after *RUN* was selected:

ZOOM IN / OUT
QUIT PROGRAM

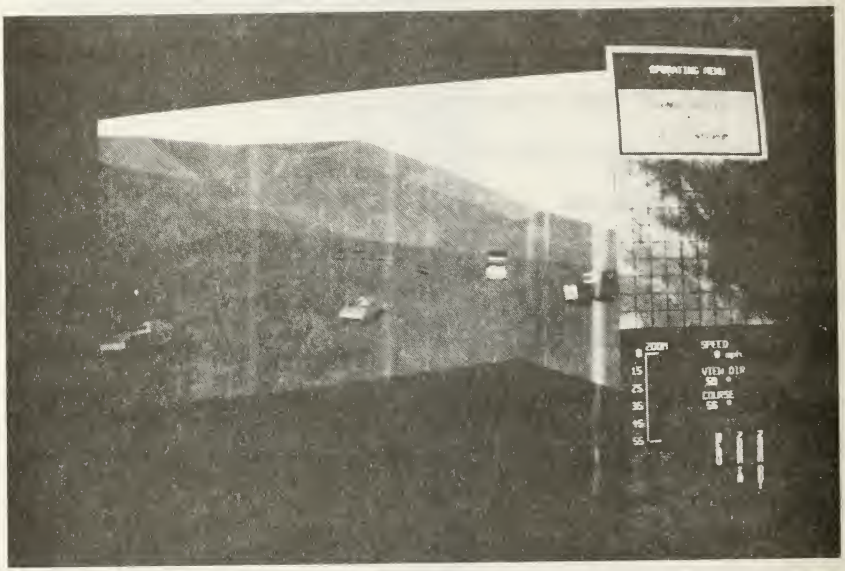


Figure 5-3. Jeep View

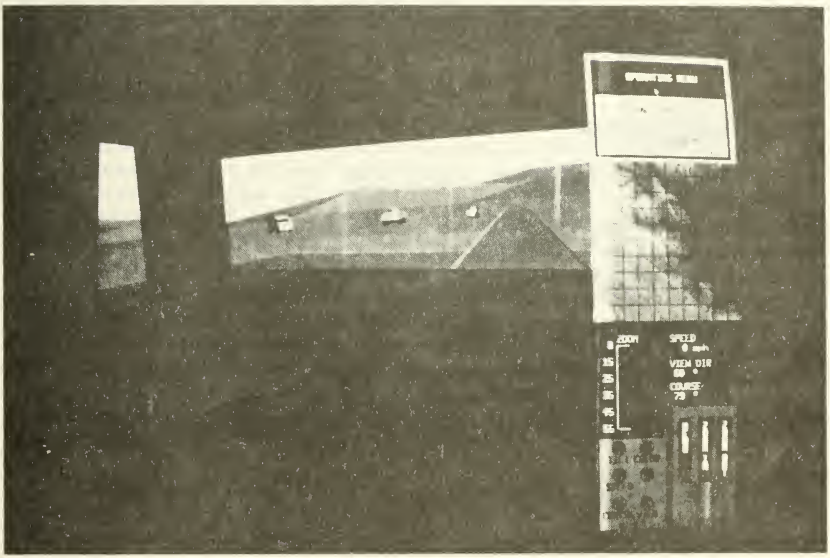


Figure 5-4. Tank View

The viewing vehicle can be changed by moving the cursor over any vehicle ICON and pushing the left mouse button. The driving display then appears with the view from the newly selected vehicle.

If *QUIT* is selected, the display changes to the large contour map without any vehicle ICONS, with menu selections identical to what is presented prior to selecting *RUN*. Now the vehicles can be laid down as if just starting the program.

To stop the program, *QUIT PROGRAM* is selected. Typing control C also exits the program, saving the last frame displayed, which can be saved in a file if desired.

4. Target Destruction

When the vehicle system is networking with the missile system, at sometime a vehicle will be hit by a missile. If a non-driven vehicle is hit, an explosion appears on the screen at the same location of the hit vehicle. The large contour map is then displayed with a red X at the location of the destroyed vehicle. If the driven vehicle is hit by the missile, the screen changes into a series of flashes and then the large contour map is displayed. The driving display is restarted by selecting a vehicle that has not been destroyed.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. LIMITATIONS

The Moving Vehicle Simulator is limited in two general respects. The real-time simulation display rate cannot be improved beyond the capacity of the present hardware configuration. In addition, the networking implementation is simple and does not provide the general distributed system described in Chapter IV.

Due to hardware performance, several specific limitations are present in the Moving Vehicle Simulator with respect to display realism. First, vehicle objects displayed are constructed using special drawing techniques versus sorting routines for drawing order. This is done to maintain a fast frame rate required for a real-time display. Once the design of special hardware provides faster Z buffer and polygon fill rates, the vehicle objects can then be constructed using a sorted drawing order and made with more polygons for added detail.

Second, when a vehicle travels in a northerly direction over a hill made up of the two triangles of a single grid square, distortion occurs as it passes over the crest of the hill (Figure 6-1). This is due to a drawing order of lower triangle, then upper triangle, and finally vehicle. The vehicle draws over the upper triangle when it should be obscured by this triangle. This drawing order can be corrected by assigning each vehicle to either the upper or lower triangle of a grid square. Now the vehicles are sorted based on which section of each grid square they are in, and drawn with that section of the grid square. This is not implemented in this study based on real-time performance

Figure 6-1. Display Scene

constraints. A modification of both *update_vehicle_grid()* and *display_terrain()* would have to be made to decide in which order to draw the grid square triangles and the vehicles in this case. This is a time intensive computation which lowers the frame rate to an unacceptable level.

Third, the terrain modeled in this study has no cultural features, such as lakes, trees and bushes. All of the terrain polygons are shaded to give a checkerboard display. The checkerboard effect is not realistic. However it gives a visual effect of motion and depth when viewing or travelling on the terrain. The integration of cultural features using texture maps is under study at the Naval Postgraduate School. Presently, to include the simplest texture map, would take too long a time period for a real-time display.

Fourth, the vehicle object's are not light-shaded for all orientations on the terrain map. Each vehicle is light-shaded during program initialization for a northern light source and an eastern vehicle course. No matter what course the vehicle travels on the terrain, it is always shaded as if the sun is on its left side. This can be corrected by calling the light shading function *for each polygon of the moving vehicle,* periodically throughout the program's execution. Since each call to the *light_orient()* function takes 0.34 millisec, the refresh rate of the display would be unacceptable.

Fifth, a vehicle can drive through another vehicle causing a distorted display. The implementation of an algorithm that would decide when two vehicles occupied the same space on the terrain, would require calculating the distance between all vehicles each frame. If a collision is imminent, one of the vehicles could be turned away or stopped. The calculation of the distance between all the vehicles each frame can not be implemented in a time frame to provide a fast frame rate.

The Moving Vehicle Simulator can be operated in networked mode with the FOG-M Simulator, but only one console of each type can be included in this network. In addition, the use of blocking socket I/O as described in Chapter IV removes some of the capabilities of the stand-alone mode of the Moving Vehicle Simulator. The vehicle operator cannot decide to drive a different vehicle while the FOG-M missile is in flight, and he cannot reset the Moving Vehicle Simulator at any time during the display loop. Both of these features are available in the stand-alone mode of operation.

B. FUTURE RESEARCH

Hardware improvements will allow more cultural features to be incorporated to improve display realism without sacrificing the display update rate. Two areas could be addressed first that would lend much more realism to the display at minimal cost. A dynamic lighting model could provide such features as fog or dust or a changing set of weather conditions, and reducing the grid square dimensions would produce smoother looking terrain. More costly improvements could utilize Gouraud shading for polygon coloring and Z-Buffering for hidden surface removal. Current research at the Naval Postgraduate School is investigating the use of texture mapping in real-time displays, and the use of a LISP machine to provide path planning for vehicles in the display. Off-loading non-graphics processing to other machines, such as path planning and updating the moving platform position, speed and other attributes, would serve to increase both the simulator frame rate and the "look-and-feel" realism of driving across terrain. In addition, research is being conducted on implementing a network data server that would allow a separate node to handle all non-graphics processing.

LIST OF REFERENCES

[1] Smith, D. B. and Streyle, D. G., "An Inexpensive Real-Time Interactive Three-Dimensional Flight Simulation System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

[2] Hearn, D. and Baker, P. M., *Computer Graphics* (Prentice Hall, Englewood, New Jersey, 1986).

[3] Fuchs, H., Abram, G. D., and Grant, E. D., "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics 17* , (July 1983).

[4] *IRIS User's Manual Version 2.0* (Silicon Graphics, Inc., Mountain View, California, 1986).

[5] Leffler, S. J., Fabry, R. S., Joy, W. N., Lapsley, P., Miller, S., and Torek, C., "An Advanced 4.3BSD Interprocess Communication Tutorial," *UNIX Programmer's Supplementary Documents 1*, (1986).

Distribution List for Dr. Michael J. Zyda

Defense Technical Information Center, Cameron Station, Alexandria, VA 22314	2 copies
Library, Code 0142 Naval Postgraduate School, Monterey, CA 93943	2 copies
Center for Naval Analyses, 4401 Ford Avenue Alexandria, VA 22302-0268	1 copy
Director of Research Administration, Code 012, Naval Postgraduate School, Monterey, CA 93943	1 copy
Dr. Michael J. Zyda Naval Postgraduate School, Code 52, Dept. of Computer Science Monterey, California 93943-5100	200 copies
Mr. Bill West, HQ, USACDEC, Attention: ATEC-D, Fort Ord, California 93941	1 copy
John Maynard, Naval Ocean Systems Center, Code 402, San Diego, California 92152	1 copy
El Wells, Naval Ocean Systems Center, Code 443, San Diego, California 92152	1 copy
Roger Casey, Naval Ocean Systems Center, Code 84, San Diego, California 92152	1 copy
Dr. Al Zied, Naval Ocean Systems Center, Code 433, San Diego, California 92152	1 copy